Formal Specification of the Plutus Core Language (rev. 6; w/ types)

I. Plutus Core

Plutus Core is a typed, strict, eagerly-reduced λ calculus design to run as a transaction validation scripting language on blockchain systems. It's designed to be simple and easy to reason about using mechanized proof assistants and automated theorem provers. The grammar of the language is given in Figures 1, 2, and 3, using a modified s-expression format. As is standard in λ calculi, we have variables, λ abstractions, and application. In addition to this, there are also polymorphism-related abstraction and instantiation, data constructors, case expressions, declared names, computational primitives, primitive values, and built-in functions. Terms live within top level declarations, which can also consist of data and type declarations as well as type signature declarations. Declarations themselves reside within modules, and a program is a collection of such modules.

In this grammar, we have multi-argument application, both in types ($\texttt{[}T\ T^*\texttt{]}$) and in terms ($\texttt{[}M\ M^*\texttt{]}$). This is to be understood as a convenient form of syntactic sugar for iterated binary application, and below we will treat only the binary case.

As an example, consider the program in Figure 4, which defines the type of natural numbers as well as lists, and the factorial and map functions. This program is not the most readable, which is to be expected from a representation intended for machine interpretation rather than human interpretation, but it does make explicit precisely what the roles are of the various parts.

II. 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 5, we define the grammar ofthe various kinds of contexts that these judgments hold under. Nominal contexts contain information about the various declared names that exist within the system — module names, term names and their definitions, type and value constructors, and type names, along with information about whether they're exported or not. Variable contexts contain information about the nature of variables — type variables weith their kind, and term variables with their type.

Then, in Figure 6, we define what it means for a type to inhabit a kind. Plutus Core is a higher-kinded version of System-F with constructors and some primitive types, so we have a number of standard System-F rules together with some obvious extensions

Next, in Figure 7, we define the type checking judgment that explains when a type contains a term. This is defined together with Figure 8's type synthesis judgment, which explains how a term synthesizes a type. Together, these two judgments constitute a standard bidirectional type theory.

A number of auxiliary judgments are defined in Figure 9. Finally, we define the various elaboration judgments in Figure 10, which explain how declarations, modules, and programs elaborate out to complete nominal contexts. The overall structure of the system is therefore viewable as a method of transforming a program into a collection of type and term declarations involving well-formed types and terms.

Tm	M	::=	x	variable
			qn	declared name
			$\texttt{(isa}\ M\ T\texttt{)}$	type annotation
			$\texttt{(abs}\ x\ M\texttt{)}$	type abstraction
			$\texttt{(inst}\ M\ T\texttt{)}$	type instantiation
			$\texttt{(lam}\ x\ M\texttt{)}$	λ abstraction
			$\texttt{[}M\ M^+\texttt{]}$	function application
			$\texttt{(con}\ qc\ M^*\texttt{)}$	constructed data
			$\texttt{(case}\ M\ C^*\texttt{)}$	case
			$\texttt{(success}\ M\texttt{)}$	success
			$\texttt{(failure)}$	failure
			$\texttt{(txhash)}$	transaction hash
			$\texttt{(blocknum)}$	block number
			$\texttt{(blocktime)}$	block time
			$\texttt{(bind}\ M\ x\ M\texttt{)}$	computation bind
			i	primitive integer
			f	primitive float
			b	primitive bytestring
			$\texttt{(builtin}\ n\ M^*\texttt{)}$	built-in function
Cl	C	::=	$\texttt{(}qc\ \texttt{(}x^*\texttt{)}\ M\texttt{)}$	case clause
Prg	G	::=	$\texttt{(program}\ L^*\texttt{)}$	program
Mod	L	::=	$\texttt{(module}\ l\ id\ ed\ D^*\texttt{)}$	module
ImpD	id	::=	$\texttt{(imported}\ l^*\texttt{)}$	import decls
ExpD	ed	::=	$\texttt{(exported}\ \texttt{(}tx^*\texttt{)}\ \texttt{(}n^*\texttt{))}$	export decls
Dec	D	::=	dd	data decl.
			td	type decl.
			md	term decl.
			df	term defn.
Ty	T	::=	x	type variable
			$\texttt{(fun}\ T\ T\texttt{)}$	function type
			$\texttt{(con}\ qc\ T^*\texttt{)}$	type constructor
			$\texttt{(comp}\ T\texttt{)}$	computation type
			$\texttt{(forall}\ x\ K\ T\texttt{)}$	polymorphic type
			$\texttt{bytestring}$	bytestring
			$\texttt{integer}$	integer
			$\texttt{float}$	float
			$\texttt{(lam}\ x\ K\ T\texttt{)}$	type abstraction
			$\texttt{[}T\ T^+\texttt{]}$	type application

Fig. 1. Grammar of Plutus Core

Ki	J	::=	`type`	type kind
			`(fun` K K`)`	arrow kind
TExp	tx	::=	n	type export
			`(`c `(`c^*`))`	data export
DDec	dd	::=	`(data` c `(`ks^*`)` alt^*`)`	data decl.
TDec	td	::=	`(type` n TV`)`	type decl.
KSig	ks	::=	`(`x K`)`	kind signature
Alt	alt	::=	`(`c T^*`)`	alternative
MDec	md	::=	`(declare` n T`)`	term decl.
Def	df	::=	`(define` n V`)`	name definition
Val	V	::=	`(lam` x M`)`	λ abstraction
			`(con` qc V^*`)`	constructed data
			`(success` V`)`	success
			`(failure)`	failure
			`(txhash)`	transaction hash
			`(blocknum)`	block number
			`(blocktime)`	block time
			`(bind` V x M`)`	computation bind
			i	primitive integer
			f	primitive float
			b	primitive bytestring
TyVal	TV	::=	x	type variable
			`(fun` TV TV`)`	function type
			`(con` qc TV^*`)`	type constructor
			`(comp` TV`)`	computation type
			`(lam` x K T`)`	type abstraction
			`(forall` x K T`)`	polymorphism
			`bytestring`	bytestring
			`integer`	integer
			`float`	float

Fig. 2. Grammar of Plutus Core (cont.)

QualN	qn	::=	$l.n$	qualified name
QualC	qc	::=	$l.c$	qualified constructor
Name	n	::=	[a-z][a-zA-Z0-9_']*	name
Mod	l	::=	[A-Z][a-zA-Z0-9_']*	module name
Con	c	::=	[A-Z][a-zA-Z0-9_']*	constructor name
Integer	i	::=	$[+-]^?$[0-9]$^+$	integer
Float	f	::=	$[+-]^?$[0-9]$^+$(\\.[0-9]$^+ e^? \| e$)	float
Exp	e	::=	[eE]$[+-]^?$[0-9]$^+$	exponent
ByStr	b	::=	#([a-fA-F0-9][a-fA-F0-9])$^+$	hex string
			#"$char^*$"	ASCII string
Var	x	::=	[a-z][a-zA-Z0-9_']*	variable

Fig. 3. Lexical Grammar of Plutus Core

We note that the judgment for modules defines Δ'' as the resulting of extending with exports from ed. By this we mean simply that we add appropriate context judgments to mark the parts of the export declaration as being exported. The details are verbose, boring, and obvious, and so are elided. We note also that various mentions of freshness are made. The details of this are also verbose, boring, and obvious, and thus similarly elided.

III. Reduction and Execution

The execution of a program in Plutus Core does not in itself result in any reduction. Instead, the declarations are bound to their appropriate names in a declaration environment δ, which we will represent by a list of items of the form $\mathtt{n} \mapsto M$. Then, designated names can be chosen to be reduced in this declaration environment generated. For instance, we might designate the name `main` to be the name who's definition we reduce, as is done in Haskell. Declaration environments are generated by restricting nominal environments to just the term definition judgments as shown in Figure 11.

```
(program
  (module Ex
    (imported Prelude)
    (exported (...) (...))
    (data Nat () (Zero ) (Suc (con Nat )))
    (data List ((x type)) (Nil ) (Cons x (con List x)))
    (declare fibonacci (fun (con Nat ) (con Nat )))
    (define fibonacci
      (lam n
        (case (builtin equalsInteger n 0)
          (Prelude.True () 1)
          (Prelude.False ()
            (builtin multiplyInteger
              n
              [Ex.fibonacci
                (builtin subtractInteger n 1)]))))))
    (declare map
      (forall a type (forall b type
        (fun (fun a b) (fun (con List a) (con List b))))))
    (define map
      (abs a (abs b
        (lam f
          (lam xs
            (case xs
              (Ex.Nil () (con Ex.Nil))
              (Ex.Cons (x xs')
                (con Ex.Cons
                  [f x]
                  [(inst (inst Ex.map a) b) f xs']))))))))
```

Fig. 4. Example with Fibonacci and Map

To give the computation rules for Plutus Core, we must define what the return values are of the language, as given in Figure 12. Rather than using values directly, we wrap them in a return value form, because reduction steps can fail. These then let us define a parameterized binary relation $M \rightarrow^*_\delta R$ which means M eagerly reduces to R using declarations δ, in Figure 18. This uses a standard contextual dynamics to separate the local reductions, reduction contexts, and repeated reductions into separate judgments. We also define a step-indexed dynamics $M \rightarrow^n_\delta R$, which means that M reduces to R using δ in at most n steps. Step-indexed reduction is useful in settings where we want to limit the number of computational

NomCtx	Δ	::=	ϵ	empty nom. context
			Δ, ν	non-empty nom. context
Nom	ν	::=	l mod	module name
			l exptype $(n\|c)$	exported type name/con.
			l expterm $(n\|c)$	exported term name/con.
			l term $n : TV$	term name
			l def $n = V$	term definition
			l con c as $[x^*](T^*)c$	term constructor
			l tycon $c :: K^*$	type constructor
			l type $n = TV :: K$	type name
VarCtx	Γ	::=	ϵ	empty var. context
			Γ, j	non-empty var. context
CtxJ	j	::=	$x :: K$	type variable
			$x : TV$	term variable

Fig. 5. Contexts

steps that can occur. These relations represent the transitive closure of the single-step reduction relation $M \ \rightarrow_\delta \ R$, which is itself the lifting of local (i.e. β) reduction $M \ \Rightarrow_\delta \ R$ to the non-local setting by digging through a reduction context.

One such setting for step-indexing is that of blockchain transactions, for which Plutus Core has been explicitly designed. 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 systemn, 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 reductions of terms M by computing which R is such that $M \ \rightarrow_\delta^{max} \ R$.

Because built-in reduction is implemented directly in terms of meta-language functionality, the specifications for them are subtly different than for other parts of this spec. In particular, we must explain what these meta-language implementations are that constitute the implicit spec. Primitive numeric integers are implemented as Haskell $Integer$s, primitive floats as $Float$, and primitive bytestrings as $ByteString$. For numeric built-ins, the operations are interpreted as the corresponding Haskell operations. So for example, $addInteger$ is interpreted as $(+) :: Integer \rightarrow Integer \rightarrow Integer$. The function names in the definitions are the same as the Haskell implementations where applicable. Some minor differences exist in some places, however. The cryptographic functions $sha2\_256$ and $sha3\_256$, in particular. They are implemented in terms of hashing into the $SHA256$ and $SHA3256$ digest types using the $Crypto.Hash$ and $Crypto.Sign.Ed25519$ modules. More indirectly, the specification for these are the cryptographic standards for SHA2 256 and SHA3 256.

All of these operations are given in tabular form. The arguments column specifies what sorts of arguments are required for correct application of the given built in, which results in the production of an `(ok` V`)` return value that wraps the value given in the result column. When the arguments are not of the specified form, the result of the built in application is `err`.

A final note on built-in reduction is that some built-ins return constructed data using qualified names in the $Prelude$ module. This specification assumes that an implementation

$\boxed{\Delta;\Gamma \vdash_{l,\overline{l'}} \ T :: K}$

T is a type in module l of kind K, with names in nominal context Δ and variables in variable context Γ

$$\frac{\Gamma \ \ni \ x :: K}{\Delta;\Gamma \vdash_{l,\overline{l'}} \ x :: K}$$

$$\frac{\Delta \text{ permits type } n :: K \text{ in } l, \overline{l'}}{\Delta;\Gamma \vdash_{l,\overline{l'}} \ n :: K}$$

$$\frac{\Delta;\Gamma \vdash_{l,\overline{l'}} \ T :: \texttt{type} \qquad \Delta;\Gamma \vdash_{l,\overline{l'}} \ T' :: \texttt{type}}{\Delta;\Gamma \vdash_{l,\overline{l'}} \ (\texttt{fun}\ T\ T') :: \texttt{type}}$$

$$\frac{\begin{array}{c}\Delta \text{ permits type constructor } qc :: \overline{K} \text{ in } l, \overline{l'} \\ |\overline{K}| = |\overline{T}| \\ \forall i(\Delta;\Gamma \vdash_{l,\overline{l'}} \ T_i :: K_i)\end{array}}{\Delta;\Gamma \vdash_{l,\overline{l'}} \ (\texttt{con}\ qc\ \overline{T}) :: \texttt{type}}$$

$$\frac{\Delta;\Gamma \vdash_{l,\overline{l'}} \ T :: \texttt{type}}{\Delta;\Gamma \vdash_{l,\overline{l'}} \ (\texttt{comp}\ T) :: \texttt{type}}$$

$$\frac{\Delta;\Gamma, x :: K \vdash_{l,\overline{l'}} \ T :: \texttt{type}}{\Delta;\Gamma \vdash_{l,\overline{l'}} \ (\texttt{forall}\ x\ K\ T) :: \texttt{type}}$$

$$\frac{}{\Delta;\Gamma \vdash_{l,\overline{l'}} \ \texttt{integer} :: \texttt{type}}$$

$$\frac{}{\Delta;\Gamma \vdash_{l,\overline{l'}} \ \texttt{float} :: \texttt{type}}$$

$$\frac{}{\Delta;\Gamma \vdash_{l,\overline{l'}} \ \texttt{bytestring} :: \texttt{type}}$$

$$\frac{\Delta;\Gamma, x :: K \vdash_{l,\overline{l'}} \ T :: K'}{\Delta;\Gamma \vdash_{l,\overline{l'}} \ (\texttt{lam}\ x\ K\ T) :: (\texttt{fun}\ K\ K')}$$

$$\frac{\Delta;\Gamma \vdash_{l,\overline{l'}} \ T :: (\texttt{fun}\ K\ K') \qquad \Delta;\Gamma \vdash_{l,\overline{l'}} \ T' :: K}{\Delta;\Gamma \vdash_{l,\overline{l'}} \ [T\ T'] :: K'}$$

Fig. 6. Type Well-formedness

will have such a module defined, that it declares exported constructors names $True$ and $False$, and that it will always incorporated it as part of any use of Plutus Core usage so that the results of these built-ins can be used by programs in case expressions.

Moving to execution, the computation constructions `(success` M`)`, `(failure)`, `(txhash)`, `(blocknum)`, `(blocktime)`, and `(bind` M `x` M`)` constitute a first order representation of a reader monad with failure and a particular environment type. Reduction of such terms proceeds as any first order data does. However, such data can also be *executed*, which involves performing actual reader operations as well as failing. We can make an analogy to Haskell's IO, where an IO value is just a value, but certain designated names with IO type, in additional to being reduced, are also executed by the run time system. We therefore also define a binary relation $M \ \rightsquigarrow_{E,\delta}^* \ R$ that specifies when a term M reduces to return value R in some reader environment E and declaration

$$\boxed{\Delta;\Gamma \vdash_{l,\overline{l'}}\ TV \ni M}$$

Reduced type TV checks term M in module l importing $\overline{l'}$, with names in nominal context Δ and variables in variable context Γ

$$\frac{\Delta;\Gamma, x :: k \vdash_{l,\overline{l'}}\ T \ni M}{\Delta;\Gamma \vdash_{l,\overline{l'}}\ \texttt{(forall}\ x\ k\ T\texttt{)} \ni \texttt{(abs}\ x\ M\texttt{)}}$$

$$\frac{\Delta;\Gamma, x : T \vdash_{l,\overline{l'}}\ T' \ni M}{\Delta;\Gamma \vdash_{l,\overline{l'}}\ \texttt{(fun}\ T\ T'\texttt{)} \ni \texttt{(lam x}\ M\texttt{)}}$$

$$\frac{\begin{array}{c}\Delta \text{ permits constructor } qc' \text{ as } [x](\overline{T'})qc'' \text{ in } l, \overline{l'} \\ qc = qc'' \\ |\overline{M}| = |\overline{T'}| \\ \forall i([\overline{T}/\overline{x}]T'_i\ \rightarrow^*_{ty}\ TV'_i \text{ and } \Delta;\Gamma \vdash_{l,\overline{l'}}\ TV'_i \ni M_i)\end{array}}{\Delta;\Gamma \vdash_{l,\overline{l'}}\ \texttt{(con}\ qc\ \overline{T}\texttt{)} \ni \texttt{(con}\ qc'\ \overline{M}\texttt{)}}$$

$$\frac{\Delta;\Gamma \vdash_{l,\overline{l'}}\ T \ni M}{\Delta;\Gamma \vdash_{l,\overline{l'}}\ \texttt{(comp}\ T\texttt{)} \ni \texttt{(success}\ M\texttt{)}}$$

$$\frac{}{\Delta;\Gamma \vdash_{l,\overline{l'}}\ \texttt{(comp}\ T\texttt{)} \ni \texttt{(failure)}}$$

$$\frac{\Delta;\Gamma \vdash_{l,\overline{l'}}\ M \in T \qquad \Gamma\ \vdash\ T = T'}{\Delta;\Gamma \vdash_{l,\overline{l'}}\ T' \ni M}$$

Fig. 7. Type Checking

environment δ, as well as a step indexed variant $M\ \rightsquigarrow^n_{E,\delta}\ R$. The reader environment E consists of three values, a bytestring E_{txhash} which is the hash of the host transaction, an integer $E_{blocknum}$ for the block number of the host block, and an integer $E_{blocktime}$ for the block time of the host block.

Note that the success and failure terms are not effectful. That is to say, `(failure)` does not throw an exception of any sort. They are merely primitive values that represent computational success and failure. They are analogous to Haskell `Maybe` values, except that they cannot be inspected, and all computational control is done via the *bind* construct.

IV. Basic Validation Program Structure

The basic way that validation is done in Plutus Core is slightly different than in Bitcoin Script. Whereas in Bitcoin Script, a validation is successful if the validating script successfully executes and leads $true$ on the top of the stack, in Plutus Core, we have special data constructs for validation. In particular, the `(success` V`)` and `(failure)`. Any program which validates a transaction must declare a function `Validator.validator`, while the corresponding program supplied by the redeemer must declare `Redeemer.redeemer`. The declarations of both are combined into a single set of declarations, and these two declared terms are then composed with a bind. The overall validation, therefore, involves reducing the term

```
(bind Redeemer.redeemer x [Validator.validator x])
```

. If this executes to produce `(ok` V`)` for some V, then the transaction is valid, analogous to Bitcoin Script successfully executing and leaving $true$ on the top of stack. On the other hand, if it reduces to `err`, then the transaction is invalid, analogous to Bitcoin Script either leaving $false$ on the top of stack, or failing to execute. The value returned in the success case is irrelevant to validation but may be used for other purposes.

$\boxed{\Delta;\Gamma \vdash_{l,\overline{l'}} \; M \in T}$

Term M in module l importing $\overline{l'}$ synthesizes type T, with names in nominal context Δ and variables in variable context Γ

$$\frac{\Gamma \ni x : T}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; x \in T}$$

$$\frac{\Delta \text{ permits } qn : T \text{ in } l,\overline{l'}}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; qn \in T}$$

$$\frac{\Delta;\Gamma \vdash_{l,\overline{l'}} \; T \ni M}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; (\texttt{isa}\ M\ T) \in T}$$

$$\frac{\Delta;\Gamma \vdash_{l,\overline{l'}} \; M \in (\texttt{forall}\ x\ K\ T) \qquad \Delta;\Gamma \vdash_{l,\overline{l'}} \; T' :: K}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; (\texttt{inst}\ M\ T') \in [T'/x]T}$$

$$\frac{\Delta;\Gamma \vdash_{l,\overline{l'}} \; M \in (\texttt{fun}\ T\ T') \qquad \Delta;\Gamma \vdash_{l,\overline{l'}} \; T \ni M}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; [M\ N] \in T'}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash_{l,\overline{l'}} \; M \in T' \\ T' \rightarrow^*_{ty} TV \\ \overline{C} \text{ has no repeated constructors} \\ \overline{C} \text{ covers all } TV \text{ constructors} \\ \forall i(\Delta;\Gamma \vdash_{l,\overline{l'}} \; TV \ni C_i \text{ clause} \in T)\end{array}}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; (\texttt{case}\ M\ \overline{C}) \in T}$$

$$\frac{}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; (\texttt{txhash}) \in (\texttt{comp bytestring})}$$

$$\frac{}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; (\texttt{blocknum}) \in (\texttt{comp integer})}$$

$$\frac{}{\begin{array}{l}\Delta;\Gamma \vdash_{l,\overline{l'}} (\texttt{blocktime}) \\ \quad \in (\texttt{comp}\ (\texttt{con Prelude.DateTime}\ ))\end{array}}$$

$$\frac{\Delta;\Gamma \vdash_{l,\overline{l'}} \; M \in (\texttt{comp}\ T) \qquad \Delta;\Gamma, x : T \vdash_{l,\overline{l'}} \; M' \in (\texttt{comp}\ T')}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; (\texttt{bind}\ M\ \texttt{x}\ M') \in (\texttt{comp}\ T')}$$

$$\frac{}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; i \in \texttt{integer}}$$

$$\frac{}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; f \in \texttt{float}}$$

$$\frac{}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; b \in \texttt{bytestring}}$$

Fig. 8. Type Synthesis

$\boxed{\Delta;\Gamma \vdash_{l,\overline{l'}} \; TV \ni C \text{ clause} \in T}$

Type TV permits clause C in module l importing $\overline{l'}$ to be well-formed synthesizing body type T, with names in nominal context Δ and variables in variable context Γ

$$\frac{\begin{array}{c}\Delta \text{ permits constructor } qc \text{ as } [\overline{x'}](\overline{T'})qc'' \text{ in } l,\overline{l'} \\ qc' = qc'' \\ |\overline{x}| = |\overline{T'}| \\ \Delta;\Gamma,\overline{x : [\overline{TV}/\overline{x'}]T'} \vdash_{l,\overline{l'}} \; M \in T\end{array}}{\Delta;\Gamma \vdash_{l,\overline{l'}} \; (\texttt{con}\ qc'\ \overline{TV}) \ni (qc\ (\overline{x})\ M) \text{ clause} \in T}$$

$\boxed{\Delta \text{ permits type } qn :: K \text{ in } l,\overline{l'}}$

Nominal context Δ permits the use of qualified type name qn at kind K, in module l importing $\overline{l'}$

$$\frac{l = l' \qquad \Delta \ni l \text{ type } n = TV :: K}{\Delta \text{ permits type } l.n :: K \text{ in } l',\overline{l''}}$$

$$\frac{\overline{l''} \ni l' \qquad \Delta \ni l \text{ type } n = TV :: K \qquad \Delta \ni l \text{ exptype } n}{\Delta \text{ permits type } l.n :: K \text{ in } l',\overline{l''}}$$

$\boxed{\Delta \text{ permits type constructor } qc :: \overline{K} \text{ in } l,\overline{l'}}$

Nominal context Δ permits the use of qualified type constructor name qc with parameter kinds $\overline{K}$, in module l importing $\overline{l'}$

$$\frac{l = l' \qquad \Delta \ni l \text{ tycon } c :: \overline{K}}{\Delta \text{ permits type constructor } l.c :: \overline{K} \text{ in } l',\overline{l''}}$$

$$\frac{\overline{l''} \ni l \qquad \Delta \ni l \text{ tycon } c :: \overline{K} \qquad \Delta \ni l \text{ exptype } c}{\Delta \text{ permits type constructor } l.c :: \overline{K} \text{ in } l',\overline{l''}}$$

$\boxed{\Delta \text{ permits } qn : T \text{ in } l,\overline{l'}}$

Nominal context Δ permits the use of qualified name qn at type T in module l importing $\overline{l'}$

$$\frac{l = l' \qquad \Delta \ni l \text{ term } n : T}{\Delta \text{ permits } l.n : T \text{ in } l',\overline{l''}}$$

$$\frac{\overline{l''} \ni l \qquad \Delta \ni l \text{ term } n : T \qquad \Delta \ni l \text{ expterm } n}{\Delta \text{ permits } l.n : T \text{ in } l',\overline{l''}}$$

$\boxed{\Delta \text{ permits constructor } qc \text{ as } [\overline{x}](\overline{T})qc' \text{ in } l,\overline{l'}}$

Nominal context Δ permits the use of qualified constructor name qc with type parameters $\overline{x}$, argument types $\overline{T}$, and return type constructor qc', in module l importing $\overline{l'}$

$$\frac{l = l' \qquad \Delta \ni l \text{ con } c \text{ as } [\overline{x}](\overline{T})c'}{\Delta \text{ permits constructor } l.c \text{ as } [\overline{x}](\overline{T})l.c' \text{ in } l',\overline{l''}}$$

$$\frac{\overline{l''} \ni l \qquad \Delta \ni l \text{ con } c \text{ as } [\overline{x}](\overline{T})c' \qquad \Delta \ni l \text{ expterm } c}{\Delta \text{ permits constructor } l.c \text{ as } [\overline{x}](\overline{T})l.c' \text{ in } l',\overline{l''}}$$

Fig. 9. Auxiliary Judgments

$\boxed{G \text{ program } \dashv \Delta}$

Program G elaborates to nominal context Δ

$$\frac{\vdash \overline{L} \text{ module } \dashv \Delta}{(\texttt{program}\ \overline{L}) \text{ program } \dashv \Delta}$$

$\boxed{\Delta \vdash L \text{ module } \dashv \Delta'}$

Module L elaborates nominal context Δ to nominal context Δ'

$$\frac{\Delta \not\ni l \text{ mod} \quad \Delta \vdash_{l,\overline{l'}} \overline{D} \text{ decl} \dashv \Delta' \quad \Delta'' \text{ is } \Delta' \text{ extended with exports for everything in } ed}{\Delta \vdash (\texttt{module}\ l\ (\texttt{imported}\ \overline{l'})\ ed\ \overline{D}) \text{ module } \dashv \Delta''}$$

$\boxed{\Delta \vdash_{l,\overline{l'}} D \text{ decl} \dashv \Delta'}$

Declaration D in module l, with imported modules $\overline{l'}$, elaborates nominal context Δ to nominal context Δ'

$$\frac{c \text{ is a fresh type constructor in } l \text{ relative to } \Delta \quad \Delta \vdash_{l,\overline{l'}} \overline{alt} \text{ alt } c' \text{ on } \overline{K} \dashv \Delta'}{\Delta \vdash_{l,\overline{l'}} (\texttt{data}\ c\ (\overline{K})\ \overline{alt}) \text{ decl} \dashv \Delta'}$$

$$\frac{n \text{ is a fresh type name in } l \text{ relative to } \Delta \quad \Delta; \epsilon \vdash_{l,\overline{l'}} TV :: K}{\Delta \vdash_{l,\overline{l'}} (\texttt{type}\ n\ TV) \text{ decl} \dashv \Delta, l \text{ type } n = TV :: K}$$

$$\frac{n \text{ is a fresh term name in } l \text{ relative to } \Delta \quad \Delta; \epsilon \vdash_{l,\overline{l'}} TV :: \texttt{type}}{\Delta \vdash_{l,\overline{l'}} (\texttt{declare}\ n\ TV) \text{ decl} \dashv \Delta, l \text{ term } n : TV}$$

$$\frac{\Delta \ni l \text{ term } n : TV \qquad \Delta; \epsilon \vdash_{l,\overline{l'}} TV \ni V}{\Delta \vdash_{l,\overline{l'}} (\texttt{define n}\ V) \text{ decl} \dashv \Delta, l \text{ def } n = V}$$

$\boxed{\Delta \vdash_{l,\overline{l'}} alt \text{ alt } c \text{ on } \overline{ks} \dashv \Delta'}$

Constructor alternative alt for type constructor c with kind signatures $\overline{ks}$ in module l importing $\overline{l'}$ elaborates nominal context Δ to nominal context Δ'

$$\frac{c \text{ is a fresh constructor for } c' \text{ in } l \text{ relative to } \Delta \quad \forall i (\Delta; \overline{x :: K} \vdash_{l,\overline{l'}} T_i :: \texttt{type})}{\Delta \vdash_{l,\overline{l'}} (c\ \overline{T}) \text{ alt } c' \text{ on } \overline{(x\ K)} \dashv \Delta, l \text{ con } c \text{ as } [\overline{x}](\overline{T})c'}$$

Fig. 10. Elaboration Judgments

$$\begin{array}{lcr}
\lfloor \epsilon \rfloor & = & \epsilon \\
\lfloor \Delta, l \text{ mod} \rfloor & = & \lfloor \Delta \rfloor \\
\lfloor \Delta, l \text{ exptype } (n|c) \rfloor & = & \lfloor \Delta \rfloor \\
\lfloor \Delta, l \text{ expterm } (n|c) \rfloor & = & \lfloor \Delta \rfloor \\
\lfloor \Delta, l \text{ term } n : TV \rfloor & = & \lfloor \Delta \rfloor \\
\lfloor \Delta, l \text{ def } n = V \rfloor & = & \lfloor \Delta \rfloor, n \mapsto V \\
\lfloor \Delta, l \text{ con } c \text{ as } [\overline{x}](\overline{T})c' \rfloor & = & \lfloor \Delta \rfloor \\
\lfloor \Delta, l \text{ tycon } c :: \overline{K} \rfloor & = & \lfloor \Delta \rfloor \\
\lfloor \Delta, l \text{ type } n = TV :: K \rfloor & = & \lfloor \Delta \rfloor
\end{array}$$

Fig. 11. Declaration Environment Generation

$$\begin{array}{llll}
\text{Ret} & R & ::= & (\texttt{ok}\ M) & \text{returned value} \\
 & & & \texttt{err} & \text{error}
\end{array}$$

Fig. 12. Return Values of Plutus Core

$$\begin{array}{llll}
\text{TCtx} & TE & ::= & \circ & \text{hole} \\
 & & & (\texttt{fun}\ TE\ T) & \text{left arrow} \\
 & & & (\texttt{fun}\ TV\ TE) & \text{right arrow} \\
 & & & (\texttt{con}\ qc\ TV^*\ TE\ T^*) & \text{type constructor} \\
 & & & (\texttt{comp}\ TE) & \text{computation} \\
 & & & (\texttt{forall}\ x\ K\ TE) & \text{forall} \\
 & & & (\texttt{fun}\ TE\ T) & \text{left app} \\
 & & & (\texttt{fun}\ TV\ TE) & \text{right app}
\end{array}$$

Fig. 13. Grammar of Type Reduction Contexts

$$\begin{aligned}
\circ\{T\} &= T \\
(\texttt{fun}\ TE\ T')\{T\} &= (\texttt{fun}\ TE\{T\}\ T') \\
(\texttt{fun}\ TV\ TE)\{T\} &= (\texttt{fun}\ TV\ TE\{T\}) \\
(\texttt{con}\ qc\ \vec{TV}\ TE\ \vec{T'})\{T\} &= (\texttt{con}\ qc\ \vec{TV}\ TE\{T\}\ \vec{T'}) \\
(\texttt{comp}\ TE)\{T\} &= (\texttt{comp}\ TE\{T\}) \\
(\texttt{forall}\ x\ k\ TE)\{T\} &= (\texttt{forall}\ x\ k\ TE\{T\}) \\
[TE\ T']\{T\} &= [TE\{T\}\ T'] \\
[TV\ TE]\{T\} &= [TV\ TE\{T\}]
\end{aligned}$$

Fig. 14. Type Context Insertion

$\boxed{T \ \rightarrow^*_{ty} \ TV}$

Type T reduces to type value TV in some number of steps

$$\frac{}{TV \ \rightarrow^*_{ty} \ TV}$$

$$\frac{T \ \rightarrow_{ty} \ T' \qquad T' \ \rightarrow^*_{ty} \ TV}{T \ \rightarrow^*_{ty} \ TV}$$

$\boxed{T \ \rightarrow_{ty} \ T'}$

Type T reduces in one step to type T'

$$\frac{T \ \Rightarrow_{ty} \ T'}{TE\{T\} \ \rightarrow_{ty} \ TE\{T'\}}$$

$\boxed{T \ \Rightarrow_{ty} \ T'}$

Type T locally reduces to type T'

$$\frac{}{\texttt{[(lam}\ x\ k\ T\texttt{)}\ T'\texttt{]} \ \Rightarrow_{ty} \ [T'/x]T}$$

Fig. 15. Type Reduction via Contextual Dynamics

Ctx	E	$::=$	$\circ$	hole
			$\texttt{[}E\ M\texttt{]}$	left app
			$\texttt{[}V\ E\texttt{]}$	right app
			$\texttt{(con}\ qc\ V^*\ E\ M^*\texttt{)}$	condata
			$\texttt{(case}\ E\ C^*\texttt{)}$	case
			$\texttt{(success}\ E\texttt{)}$	success
			$\texttt{(bind}\ E\ x\ M\texttt{)}$	bind
			$\texttt{(builtin}\ n\ V^*\ E\ M^*\texttt{)}$	builtin

Fig. 16. Grammar of Reduction Contexts

$$\begin{aligned}
\circ\{N\} &= N \\
\texttt{[}E\ M\texttt{]}\{N\} &= \texttt{[}E\{N\}\ M\texttt{]} \\
\texttt{[}M\ E\texttt{]}\{N\} &= \texttt{[}M\ E\{N\}\texttt{]} \\
\texttt{(con}\ qc\ \vec{V}\ E\ \vec{M}\texttt{)}\{N\} &= \texttt{(con}\ qc\ \vec{V}\ E\{N\}\ \vec{M}\texttt{)} \\
\texttt{(case}\ E\ \vec{C}\texttt{)}\{N\} &= \texttt{(case}\ E\{N\}\ \vec{C}\texttt{)} \\
\texttt{(success}\ E\texttt{)}\{N\} &= \texttt{(success}\ E\{N\}\texttt{)} \\
\texttt{(bind}\ E\ x\ M\texttt{)}\{N\} &= \texttt{(bind}\ E\{N\}\ x\ M\texttt{)}
\end{aligned}$$

Fig. 17. Context Insertion

$\boxed{M \ \rightarrow^*_{\delta} \ R}$

Term M reduces in some number of steps to return value R in declaration environment δ

$$\frac{}{V \ \rightarrow^*_{\delta} \ \texttt{(ok}\ V\texttt{)}}$$

$$\frac{M \ \rightarrow_{\delta} \ \texttt{(ok}\ M'\texttt{)} \qquad M' \ \rightarrow^*_{\delta} \ R}{M \ \rightarrow^*_{\delta} \ R}$$

$$\frac{M \ \rightarrow_{\delta} \ \texttt{err}}{M \ \rightarrow^*_{\delta} \ \texttt{err}}$$

$\boxed{M \ \rightarrow_{\delta} \ R}$

Term M reduces in one step to return value R in declaration environment δ

$$\frac{M \ \Rightarrow_{\delta} \ \texttt{(ok}\ M'\texttt{)}}{E\{M\} \ \rightarrow_{\delta} \ \texttt{(ok}\ E\{M'\}\texttt{)}}$$

$$\frac{M \ \Rightarrow_{\delta} \ \texttt{err}}{E\{M\} \ \rightarrow_{\delta} \ \texttt{err}}$$

Fig. 18. Reduction via Contextual Dynamics

$\boxed{M \ \Rightarrow_{\delta} \ R}$

Term M locally reduces to return value R in declaration context δ

$$\frac{}{qn \ \Rightarrow_{\delta, qn \mapsto M} \ \texttt{(ok}\ M\texttt{)}}$$

$$\frac{}{\texttt{[(lam}\ x\ M\texttt{)}\ V\texttt{]} \ \Rightarrow_{\delta} \ \texttt{(ok}\ [V/x]M\texttt{)}}$$

$$\frac{qc, \vec{V} \ \sim \ \vec{C} \ \triangleright \ R}{\texttt{(case (con}\ qc\ \vec{V}\texttt{)}\ \vec{C}\texttt{)} \ \Rightarrow_{\delta} \ R}$$

$$\frac{n \text{ on } \vec{V} \text{ reduces to } R}{\texttt{(builtin}\ n\ \vec{V}\texttt{)} \ \Rightarrow_{\delta} \ R}$$

Fig. 19. Local Reduction

$\boxed{qc, \vec{V} \ \sim \ \vec{C} \ \triangleright \ R}$

Constructor qc with arguments $\vec{V}$ matches clauses $\vec{C}$ to produce result R

$$\frac{}{qc, \vec{V} \ \sim \ \epsilon \ \triangleright \ \texttt{err}}$$

$$\frac{qc \ = \ qc'}{qc, \vec{V} \ \sim \ \texttt{(}qc'\ \texttt{(}\vec{x}\texttt{)}\ M\texttt{)}, \vec{C} \ \triangleright \ \texttt{(ok}\ [\vec{V}/\vec{x}]M\texttt{)}}$$

$$\frac{qc \ \neq \ qc' \qquad qc, \vec{V} \ \sim \ \vec{C} \ \triangleright \ R}{qc, \vec{V} \ \sim \ \texttt{(}qc'\ \texttt{(}\vec{x}\texttt{)}\ M\texttt{)}, \vec{C} \ \triangleright \ R}$$

Fig. 20. Case Matching

Instr	I	$::=$ `(success` V`)`	success
		`(failure)`	failure
		`(txhash)`	transaction hash
		`(blocknum)`	block number
		`(blocktime)`	block time
		`(bind` $M\ x\ M$`)`	computation bind

Fig. 21. Grammar of Instructions and Return Instructions

$\boxed{M \ \rightsquigarrow^{*}_{E,\delta}\ R}$

Term M executes in some number of steps to return value R in declaration environment δ and blockchain environment E

$$\frac{M \ \rightarrow^{*}_{\delta}\ \texttt{err}}{M \ \rightsquigarrow^{*}_{E,\delta}\ \texttt{err}}$$

$$\frac{M \ \rightarrow^{*}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V \neq I}{M \ \rightsquigarrow^{*}_{E,\delta}\ \texttt{err}}$$

$$\frac{M \ \rightarrow^{*}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V = \texttt{(success}\ V'\texttt{)}}{M \ \rightsquigarrow^{*}_{E,\delta}\ \texttt{(ok}\ V'\texttt{)}}$$

$$\frac{M \ \rightarrow^{*}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V = \texttt{(failure)}}{M \ \rightsquigarrow^{*}_{E,\delta}\ \texttt{err}}$$

$$\frac{M \ \rightarrow^{*}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V = \texttt{(txhash)}}{M \ \rightsquigarrow^{*}_{E,\delta}\ \texttt{(ok}\ E_{txhash}\texttt{)}}$$

$$\frac{M \ \rightarrow^{*}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V = \texttt{(blocknum)}}{M \ \rightsquigarrow^{*}_{E,\delta}\ \texttt{(ok}\ E_{blocknum}\texttt{)}}$$

$$\frac{M \ \rightarrow^{*}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V = \texttt{(blocktime)}}{M \ \rightsquigarrow^{*}_{E,\delta}\ \texttt{(ok}\ E_{blocktime}\texttt{)}}$$

$$\frac{\begin{array}{c} M \ \rightarrow^{*}_{\delta}\ \texttt{(ok}\ V\texttt{)} \\ V = \texttt{(bind}\ V_0\ x\ M_1'\texttt{)} \\ V_0 \ \rightsquigarrow^{*}_{E,\delta}\ \texttt{err} \end{array}}{M \ \rightsquigarrow^{*}_{E,\delta}\ \texttt{err}}$$

$$\frac{\begin{array}{c} M \ \rightarrow^{*}_{\delta}\ \texttt{(ok}\ V\texttt{)} \\ V = \texttt{(bind}\ V_0\ x\ M_1'\texttt{)} \\ V_0 \ \rightsquigarrow^{*}_{E,\delta}\ \texttt{(ok}\ V'\texttt{)} \\ [V'/x]M_1' \ \rightsquigarrow^{*}_{E,\delta}\ R \end{array}}{M \ \rightsquigarrow^{*}_{E,\delta}\ R}$$

Fig. 22. Execution

$\boxed{M \ \rightsquigarrow^{n}_{E,\delta}\ R}$

Term M executes in n steps to return value R in declaration environment δ and blockchain environment E

$$\frac{M \ \rightarrow^{n}_{\delta}\ \texttt{err}}{M \ \rightsquigarrow^{n}_{E,\delta}\ \texttt{err}}$$

$$\frac{M \ \rightarrow^{n}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V \neq I}{M \ \rightsquigarrow^{n}_{E,\delta}\ \texttt{err}}$$

$$\frac{M \ \rightarrow^{n}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V = \texttt{(success}\ V'\texttt{)}}{M \ \rightsquigarrow^{n}_{E,\delta}\ \texttt{(ok}\ V'\texttt{)}}$$

$$\frac{M \ \rightarrow^{n}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V = \texttt{(failure)}}{M \ \rightsquigarrow^{n}_{E,\delta}\ \texttt{err}}$$

$$\frac{M \ \rightarrow^{n}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V = \texttt{(txhash)}}{M \ \rightsquigarrow^{n}_{E,\delta}\ \texttt{(ok}\ E_{txhash}\texttt{)}}$$

$$\frac{M \ \rightarrow^{n}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V = \texttt{(blocknum)}}{M \ \rightsquigarrow^{n}_{E,\delta}\ \texttt{(ok}\ E_{blocknum}\texttt{)}}$$

$$\frac{M \ \rightarrow^{n}_{\delta}\ \texttt{(ok}\ V\texttt{)} \qquad V = \texttt{(blocktime)}}{M \ \rightsquigarrow^{n}_{E,\delta}\ \texttt{(ok}\ E_{blocktime}\texttt{)}}$$

$$\frac{\begin{array}{c} M \ \rightarrow^{n}_{\delta}\ \texttt{(ok}\ V\texttt{)} \\ V = \texttt{(bind}\ V_0\ x\ M_1'\texttt{)} \\ V_0 \ \rightsquigarrow^{n'}_{E,\delta}\ \texttt{err} \end{array}}{M \ \rightsquigarrow^{n+n'}_{E,\delta}\ \texttt{err}}$$

$$\frac{\begin{array}{c} M \ \rightarrow^{n}_{\delta}\ \texttt{(ok}\ V\texttt{)} \\ V = \texttt{(bind}\ V_0\ x\ M_1'\texttt{)} \\ V_0 \ \rightsquigarrow^{n'}_{E,\delta}\ \texttt{(ok}\ V'\texttt{)} \\ [V'/x]M_1' \ \rightsquigarrow^{n''}_{E,\delta}\ R \end{array}}{M \ \rightsquigarrow^{n+n'+n''+1}_{E,\delta}\ R}$$

Fig. 23. Indexed Execution

Builtin Name	*Arguments*	*Result*
addInt	$i_0\ i_1$	$i_0 + i_1$
subtractInt	$i_0\ i_1$	$i_0 - i_1$
multiplyInt	$i_0\ i_1$	$i_0 \times i_1$
divideInt	$i_0\ i_1$	$div\ i_0\ i_1$
remainderInt	$i_0\ i_1$	$mod\ i_0\ i_1$
lessThanInt	$i_0\ i_1$	$i_0 < i_1$
lessThanEqualsInt	$i_0\ i_1$	$i_0 <= i_1$
greaterThanInt	$i_0\ i_1$	$i_0 > i_1$
greaterThanEqualsInt	$i_0\ i_1$	$i_0 >= i_1$
equalsInt	$i_0\ i_1$	$i_0 == i_1$
intToFloat	i	$intToFloat\ i$
intToByteString	i	$intToByteString\ i$
addFloat	$f_0\ f_1$	$f_0 + f_1$
subtractFloat	$f_0\ f_1$	$f_0 - f_1$
multiplyFloat	$f_0\ f_1$	$f_0 \times f_1$
divideFloat	$f_0\ f_1$	f_0 / f_1
lessThanFloat	$f_0\ f_1$	$f_0 < f_1$
lessThanEqualsFloat	$f_0\ f_1$	$f_0 <= f_1$
greaterThanFloat	$f_0\ f_1$	$f_0 > f_1$
greaterThanEqualsFloat	$f_0\ f_1$	$f_0 >= f_1$
equalsFloat	$f_0\ f_1$	$f_0 == f_1$
ceil	f	$ceil\ f$
floor	f	$floor\ f$
round	f	$round\ f$
concatenate	$b_0\ b_1$	$concat\ [b_0, b_1]$
take	$i\ b$	$take\ (fromIntegral\ i)\ b$
drop	$i\ b$	$drop\ (fromIntegral\ i)\ b$
sha2_256	b	$sha2\_256\ b$
sha3_256	b	$sha3\_256\ b$
equalsByteString	$b_0\ b_1$	$b_0 == b_1$

Fig. 24. Builtin Reductions