Formal Specification of the Plutus Core Language (rev. 5; w/ types)

I. Plutus Core

Plutus Core is an untyped 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 Figure 1, using an s-expression format. As is standard in λ calculi, we have variables, λ abstractions, and application. In addition to this, there are also local let bindings, data constructors which have arguments, case terms which match on term, declared names, computational primitives (success, failure, and binding), primitive values, and built-in functions. Terms live within a top-level program which consists of some declarations of names to be some value forms.

As a few examples, consider the program in Figure 4, which defines 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. Scope Correctness

We define for Plutus Core a scope correctness judgment, which explains when a program's use of scoped objects (variables and names) is valid. This judgment is defined inductively as shown in Figure **??**, with auxiliary judgments in Figure **??**. Note that scope correctness forbids shadowing of names.

Regarding notation: in these definitions, we use vector arrows, subscripts, and enclosing parentheses to denote various kinds of repetition. A vector arrow use of the form $\vec{M}$, where M is a meta-syntactic variable means multiple Ms in sequence. A subscript form M_i means the i-th element of a sequence $\vec{M}$. We also use parentheses and subscripts of the form $(M_i)_i$ to indicate the sequence formed by the various Ms. This is especially useful when the sequence is seen as being constructed by a bidirectional rule. Lastly, we use vector arrows over non-meta-syntactic variables like $\overrightarrow{(\texttt{foo } x\ y)}$ to indicate a sequence of such things, which allows us to refer to the associated sequences $\vec{x}$ and $\vec{y}$ in the above ways.

Plutus Core's module system defines two kinds of declared names. Names declared using the `exp` and `expcon` keywords are exported declaration of terms and constructors, respectively, and are usable by any other module. Names declared using the `loc` and `loccon` keywords are local, and usable only in their declaring modules. This distinction between exported and local names permits a simple form of abstraction at the module level. When scope checking a module, we also require that the declarations and the definitions match, that is to say, all declared names get defined, and all defined names are declared.

QualN	qn	::=	$l.n$	qualified name
QualC	qc	::=	$l.c$	qualified constructor
Tm	M	::=	x	variable
			qn	declared name
			$(\texttt{ann } M\ T)$	type annotation
			$(\texttt{lam } x\ M)$	λ abstraction
			$[M\ M]$	function application
			$(\texttt{con } qc\ M^*)$	constructed data
			$(\texttt{case } M\ C^*)$	case
			$(\texttt{success } M)$	success
			$(\texttt{failure})$	failure
			$(\texttt{txhash})$	transaction hash
			$(\texttt{blocknum})$	block number
			$(\texttt{blocktime})$	block time
			$(\texttt{bind } M\ x\ M)$	computation bind
			i	primitive integer
			f	primitive float
			b	primitive bytestring
			$(\texttt{builtin } n\ M^*)$	built-in function
Cl	C	::=	$(\texttt{cl } qc\ (x^*)\ M)$	case clause
Prg	G	::=	$(\texttt{program } L^*)$	program
Mod	L	::=	$(\texttt{module } l\ id\ ed\ ld\ D^*)$	module
ImpD	id	::=	$(\texttt{imported } l^*)$	import decls
ExpD	ed	::=	$(\texttt{exported } dc^*)$	export decls
LocD	ld	::=	$(\texttt{local } dc^*)$	local decls
Def	D	::=	$(\texttt{define } n\ V)$	name definition
Ty	T	::=	x	type variable
			$(\texttt{arrT } T\ T)$	function type
			$(\texttt{conT } qc\ T^*)$	type constructor
			$(\texttt{compT } T)$	computation type
			$(\texttt{forallT } x\ k\ T)$	polymorphic type
			$\texttt{bytestringT}$	bytestring
			$\texttt{integerT}$	integer
			$\texttt{floatT}$	float
			$(\texttt{lamT } x\ k\ T)$	type abstraction
			$(\texttt{appT } T\ T)$	type application

Fig. 1. Grammar of Plutus Core

Plutus Core also explicitly imports modules. This forces the declaration context to be restricted when scope checking the declarations of a given module, reducing the overall declaration context to only those entries for modules in the imported collection.

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 $\texttt{n} \mapsto M$.

Ki	k	::=	typeK	type kind
			(arrK k k)	arrow kind
Dec	dc	::=	td	type decl.
			md	term decl.
TDec	td	::=	(data c (ks^*) alt^*)	type decl.
KSig	ks	::=	(x k)	kind signature
Alt	alt	::=	(c T^*)	alternative
MDec	md	::=	(declare n T)	term decl.
CAr	ca	::=	((x^*) (T^*) qc)	constructor arity
TAr	ta	::=	(k^*)	type constructor arity
Sig	s	::=	(n TV)	type signature
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
			(arrT TV TV)	function type
			(conT qc TV^*)	type constructor
			(compT TV)	computation type
			(lamT x K T)	type abstraction
			(forallT x K T)	polymorphic type
			bytestringT	bytestring
			integerT	integer
			floatT	float

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

```
(program
  (module Ex
    (imported Prelude)
    (exported (Nil 0)  (Cons 2))fibonacci map
    (local )
    (define fibonacci
      (lam n
        (case (builtin equalsInteger n 0)
          (cl Prelude.True () 1)
          (cl Prelude.False ()
            (builtin multiplyInteger
              n
              [Ex.fibonacci
                (builtin subtractInteger n 1)])))))))
    (define map
      (lam f
        (lam xs
          (case xs
            (cl Ex.Nil ()
              (con Ex.Nil))
            (cl Ex.Cons (x xs')
              (con Ex.Cons
                [f x]
                [[Ex.map f] xs'])))))))))
```

Fig. 4.   Example with Fibonacci and Map

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

Fig. 3.   Lexical Grammar of Plutus Core

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. The generation judgment for generating this is given by the relation $P \ggg \delta$, defined in Figure 13.

To give the computation rules for Plutus Core, we must define what the return values are of the language, as given in Figure 14. 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 20. This uses a standard contextual dynamics to

NomCtx	Δ	::=	ϵ	empty nominal contex
			Δ,ν	non-empty nominal contex
Nom	ν	::=	l mod	module nam
			l loc $n : TV$	local nam
			l exp $n : TV$	exported nam
			l def $n = V$	name definitio
			l loccon c as $[x^*](T^*)c$	local constructo
			l expcon c as $[x^*](T^*)c$	exported constructo
			l loctycon $c :: \vec{k}$	local type constructo
			l exptycon $c :: \vec{k}$	exported type constructor
VarCtx	Γ	::=	ϵ	empty variable contex
			Γ,j	non-empty variable contex
CtxJ	j	::=	$x :: k$	type variabl
			$x : TV$	term variabl

Fig. 5.   Contexts

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 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.

$$\boxed{\Delta \; ; \; \Gamma \; \vdash \; T \text{ type } l :: k}$$

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

$$\frac{\Gamma \; \ni \; x :: k}{\Delta \; ; \; \Gamma \; \vdash \; x \text{ type } l :: k}$$

$$\frac{\Delta \; ; \; \Gamma \; \vdash \; T \text{ type } l :: \texttt{typeK} \qquad \Delta \; ; \; \Gamma \; \vdash \; T' \text{ type } l :: \texttt{typeK}}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{arrT } T \; T') \text{ type } l :: \texttt{typeK}}$$

$$\frac{\Delta \text{ permits type constructor } qc :: \vec{k} \text{ in } l \qquad |\vec{k}| = |\vec{T}| \qquad \forall i(\Delta \; ; \; \Gamma \; \vdash \; T_i \text{ type } l :: k_i)}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{conT } qc \; \vec{T}) \text{ type } l :: \texttt{typeK}}$$

$$\frac{\Delta \; ; \; \Gamma \; \vdash \; T \text{ type } l :: \texttt{typeK}}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{compT } T) \text{ type } l :: \texttt{typeK}}$$

$$\frac{\Delta \; ; \; \Gamma, x :: k \; \vdash \; T \text{ type } l :: \texttt{typeK}}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{forallT } x \; k \; T) \text{ type } l :: \texttt{typeK}}$$

$$\frac{}{\Delta \; ; \; \Gamma \; \vdash \; \texttt{integerT} \text{ type } l :: \texttt{typeK}}$$

$$\frac{}{\Delta \; ; \; \Gamma \; \vdash \; \texttt{floatT} \text{ type } l :: \texttt{typeK}}$$

$$\frac{}{\Delta \; ; \; \Gamma \; \vdash \; \texttt{bytestringT} \text{ type } l :: \texttt{typeK}}$$

$$\frac{\Delta \; ; \; \Gamma, x :: k \; \vdash \; T \text{ type } l :: k'}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{lamT } x \; k \; T) \text{ type } l :: (\texttt{arrK } k \; k')}$$

$$\frac{\Delta \; ; \; \Gamma \; \vdash \; T \text{ type } l :: (\texttt{arrK } k \; k') \qquad \Delta \; ; \; \Gamma \; \vdash \; T' \text{ type } l :: k}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{appT } T \; T') \text{ type } l :: k'}$$

Fig. 6. Type Well-formedness

$$\boxed{\Delta \; ; \; \Gamma \; \vdash \; TV \; \ni \; M \text{ term } l}$$

Reduced type TV checks term M in module l, with names in nominal context Δ and variables in variable context Γ

$$\frac{\Delta \; ; \; \Gamma, x : T \; \vdash \; T' \; \ni \; M \text{ term } l}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{arrT } T \; T') \; \ni \; (\texttt{lam x } M) \text{ term } l}$$

$$\frac{\begin{array}{c}\Delta \text{ permits constructor } qc' \text{ as } [\vec{x}](\vec{T'})qc'' \text{ in } l \\ qc = qc'' \\ |\vec{M}| = |\vec{T'}| \\ \forall i([\vec{T}/\vec{x}]T_i' \; \to_{ty}^* \; TV_i' \text{ and } \Delta \; ; \; \Gamma \; \vdash \; TV_i' \; \ni \; M_i \text{ term } l)\end{array}}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{conT } qc \; \vec{T}) \; \ni \; (\texttt{con } qc' \; \vec{M}) \text{ term } l}$$

$$\frac{\Delta \; ; \; \Gamma \; \vdash \; T \; \ni \; M \text{ term } l}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{compT } T) \; \ni \; (\texttt{success } M) \text{ term } l}$$

$$\frac{}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{compT } T) \; \ni \; (\texttt{failure}) \text{ term } l}$$

$$\frac{\Delta \; ; \; \Gamma, x :: k \; \vdash \; T \; \ni \; M \text{ term } l}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{forallT } x \; k \; T) \; \ni \; M \text{ term } l}$$

$$\frac{\Delta \; ; \; \Gamma \; \vdash \; M \text{ term } l \in T \qquad \Gamma \; \vdash \; T \sqsubseteq T'}{\Delta \; ; \; \Gamma \; \vdash \; T' \; \ni \; M \text{ term } l}$$

Fig. 7. Type Checking

ever. $drop(x, y)$ in the spec corresponds to the Haskell code $drop\ (fromIntegral\ x)\ y$, and similarly for $take$, simply because of how these functions are implemented for $ByteString$. The cryptographic functions $sha2\_256$ and $sha3\_256$ are also different, this time more so. 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 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

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 $M \; \to_{\delta}^{max} \; R$.

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 $Integers$, 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 $(+) :: Integer \to Integer \to Integer$. The function names in the definitions are the same as the Haskell implementations where applicable. Some minor differences exist in some places, how-

$$\boxed{\Delta \; ; \; \Gamma \; \vdash \; M \text{ term } l \; \in \; T}$$

Term M in module l synthesizes type T, with names in nominal context Δ and variables in variable context Γ

$$\frac{\Gamma \; \ni \; x : T}{\Delta \; ; \; \Gamma \; \vdash \; x \text{ term } l \; \in \; T}$$

$$\frac{\Delta \text{ permits } qn : T \text{ in } l}{\Delta \; ; \; \Gamma \; \vdash \; qn \text{ term } l \; \in \; T}$$

$$\frac{\Delta \; ; \; \Gamma \; \vdash \; T \; \ni \; M \text{ term } l}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{ann } M \; T) \text{ term } l \; \in \; T}$$

$$\frac{\Delta \; ; \; \Gamma \; \vdash \; M \text{ term } l \; \in \; (\texttt{arrT } T \; T') \quad \Delta \; ; \; \Gamma \; \vdash \; T \; \ni \; M \text{ term } l}{\Delta \; ; \; \Gamma \; \vdash \; [M \; N] \text{ term } l \; \in \; T'}$$

$$\frac{\Delta \; ; \; \Gamma \; \vdash \; M \text{ term } l \; \in \; T' \quad T' \; \rightarrow^*_{ty} \; TV \quad \vec{C} \text{ has no repeated constructors} \quad \forall i (\Delta \; ; \; \Gamma \; \vdash \; TV \; \ni \; C_i \text{ clause } l \; \in \; T)}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{case } M \; \vec{C}) \text{ term } l \; \in \; T}$$

$$\frac{}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{txhash}) \text{ term } l \; \in \; (\texttt{compT bytestringT})}$$

$$\frac{}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{blocknum}) \text{ term } l \; \in \; (\texttt{compT integerT})}$$

$$\frac{}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{blocktime}) \text{ term } l \; \in \; (\texttt{compT } (\texttt{conT Prelude.DateTime}))}$$

$$\frac{\Delta \; ; \; \Gamma \; \vdash \; M \text{ term } l \; \in \; (\texttt{compT } T) \quad \Delta \; ; \; \Gamma, x : T \; \vdash \; M' \text{ term } l \; \in \; (\texttt{compT } T')}{\Delta \; ; \; \Gamma \; \vdash \; (\texttt{bind } M \text{ x } M') \text{ term } l \; \in \; (\texttt{compT } T')}$$

$$\frac{}{\Delta \; ; \; \Gamma \; \vdash \; i \text{ term } l \; \in \; \texttt{integerT}}$$

$$\frac{}{\Delta \; ; \; \Gamma \; \vdash \; f \text{ term } l \; \in \; \texttt{floatT}}$$

$$\frac{}{\Delta \; ; \; \Gamma \; \vdash \; b \text{ term } l \; \in \; \texttt{bytestringT}}$$

Fig. 8. Type Synthesis

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 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.

$$\boxed{\Gamma \; \vdash \; TV \; \sqsubseteq \; TV'}$$

Reduced type TV is a subtype of reduced type TV' in variable context Γ

$$\frac{\Gamma \; \ni \; x :: k}{\Gamma \; \vdash \; x \; \sqsubseteq \; x}$$

$$\frac{\Gamma \; \vdash \; S' \; \sqsubseteq \; S \quad \Gamma \; \vdash \; T \; \sqsubseteq \; T'}{\Gamma \; \vdash \; (\texttt{arrT } S \; T) \; \sqsubseteq \; (\texttt{arrT } S' \; T')}$$

$$\frac{}{\Gamma \; \vdash \; (\texttt{conT } qc \; \vec{T}) \; \sqsubseteq \; (\texttt{conT } qc \; \vec{T})}$$

$$\frac{}{\Gamma \; \vdash \; (\texttt{compT } T) \; \sqsubseteq \; (\texttt{compT } T)}$$

$$\frac{}{\Gamma \; \vdash \; \texttt{integerT} \; \sqsubseteq \; \texttt{integerT}}$$

$$\frac{}{\Gamma \; \vdash \; \texttt{floatT} \; \sqsubseteq \; \texttt{floatT}}$$

$$\frac{}{\Gamma \; \vdash \; \texttt{bytestringT} \; \sqsubseteq \; \texttt{bytestringT}}$$

$$\frac{\Gamma, x :: k \; \vdash \; T \; \sqsubseteq \; T'}{\Gamma \; \vdash \; T \; \sqsubseteq \; (\texttt{forallT } x \; k \; T')}$$

$$\frac{\exists T''([T''/x]T \; \sqsubseteq \; T')}{\Gamma \; \vdash \; (\texttt{forallT } x \; k \; T) \; \sqsubseteq \; T'}$$

Fig. 9. Subtyping

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 the 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.

$$\boxed{\Delta \; ; \; \Gamma \;\vdash\; TV \;\ni\; C \text{ clause } l \;\in\; T}$$

Type TV permits clause C in module l to be well-formed synthesizing body type T, with names in nominal context Δ and variables in variable context Γ

$$\Delta \text{ permits constructor } qc \text{ as } [\vec{x'}](\vec{T'})qc'' \text{ in } l$$
$$qc' = qc''$$
$$|\vec{x}| = |\vec{T'}|$$
$$\Delta \; ; \; \Gamma, (x_i : [\vec{TV}/\vec{x'}]T'_i)_i \;\vdash\; M \text{ term } l \;\in\; T$$
$$\overline{\Delta \; ; \; \Gamma \;\vdash\; (\texttt{conT } qc' \; \vec{TV}) \;\ni\; (\texttt{cl } qc \; (\vec{x}) \; M) \text{ clause } l \;\in\; T}$$

$$\boxed{\Delta \text{ permits type constructor } qc :: \vec{k} \text{ in } l}$$

Nominal context Δ permits the use of qualified type constructor name qc with parameter kinds $\vec{k}$, in module l

$$\frac{l = l' \qquad \Delta \;\ni\; l \text{ loctycon } c :: \vec{k}}{\Delta \text{ permits type constructor } \texttt{l.c} :: \vec{k} \text{ in } l'}$$

$$\frac{\Delta \;\ni\; l \text{ exptycon } c :: \vec{k}}{\Delta \text{ permits type constructor } \texttt{l.c} :: \vec{k} \text{ in } l'}$$

$$\boxed{\Delta \text{ permits } qn : T \text{ in } l}$$

Nominal context Δ permits the use of qualified name qn at type T in module l

$$\frac{l = l' \qquad \Delta \;\ni\; l \text{ loc } n : T}{\Delta \text{ permits } \texttt{l.n} : T \text{ in } l'}$$

$$\frac{\Delta \;\ni\; l \text{ exp } n : T}{\Delta \text{ permits } \texttt{l.n} : T \text{ in } l'}$$

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

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

$$\frac{l = l' \qquad \Delta \;\ni\; l \text{ loccon } c \text{ as } [\vec{x}](\vec{T})c'}{\Delta \text{ permits constructor } \texttt{l.c} \text{ as } [\vec{x}](\vec{T})\texttt{l.c'} \text{ in } l'}$$

$$\frac{\Delta \;\ni\; l \text{ expcon } qc \text{ as } [\vec{x}](\vec{T})c'}{\Delta \text{ permits constructor } \texttt{l.c} \text{ as } [\vec{x}](\vec{T})\texttt{l.c'} \text{ in } l'}$$

Fig. 10. Auxiliary Judgments

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

Program G elaborates to nominal context Δ

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

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

Module L elaborates nominal context Δ to nominal context Δ'

$$\Delta \;\not\ni\; l \text{ mod}$$
$$\Delta' \text{ is } \Delta \text{ restricted to } \vec{l'}$$
$$\Delta' \;\vdash\; \vec{dc} \text{ expdecl } l \;\dashv\; \Delta''$$
$$\Delta'' \;\vdash\; \vec{dc'} \text{ locdecl } l \;\dashv\; \Delta'''$$
$$\Delta''' \;\vdash\; \vec{D} \text{ define } l \;\dashv\; \Delta''''$$
$$\overline{\begin{array}{c}\Delta \;\vdash\; (\texttt{module } l \; (\texttt{imported } \vec{l'}) \; (\texttt{exported } \vec{dc}) \\ (\texttt{local } \vec{dc'}) \; \vec{D}) \text{ module } \dashv \Delta''''\end{array}}$$

$$\boxed{\Delta \;\vdash\; dc \text{ expdecl } l \;\dashv\; \Delta'}$$

Export declaration dc in module l elaborates nominal context Δ to nominal context Δ'

$$\Delta \;\not\ni\; l \text{ exptycon } c :: \vec{k'}$$
$$\Delta \;\not\ni\; l \text{ loctycon } c :: \vec{k'}$$
$$\Delta, l \text{ exptycon } c :: (k_i)_i \;\vdash\; \vec{alt} \text{ expalt } c \text{ on } ((x_i \; k_i))_i \text{ in } l \;\dashv\; \Delta'$$
$$\overline{\Delta \;\vdash\; (\texttt{data } c \; (((x_i \; k_i))_i) \; \vec{alt}) \text{ expdecl } l \;\dashv\; \Delta'}$$

$$\Delta \;\not\ni\; l \text{ exp } n : T$$
$$\Delta \;\not\ni\; l \text{ loc } n : T$$
$$\Delta \; ; \;\vdash\; T \text{ type } l :: \texttt{typeK}$$
$$\overline{\Delta \;\vdash\; (\texttt{declare } n \; T) \text{ expdecl } l \;\dashv\; \Delta, l \text{ exp } n : T}$$

$$\boxed{\Delta \;\vdash\; dc \text{ locdecl } l \;\dashv\; \Delta'}$$

Local declaration dc in module l elaborates nominal context Δ to nominal context Δ'

$$\Delta \;\not\ni\; l \text{ exptycon } c :: \vec{k'}$$
$$\Delta \;\not\ni\; l \text{ loctycon } c :: \vec{k'}$$
$$\Delta, l \text{ loctycon } c :: (k_i)_i \;\vdash\; \vec{alt} \text{ localt } c \text{ on } ((x_i \; k_i))_i \text{ in } l \;\dashv\; \Delta'$$
$$\overline{\Delta \;\vdash\; (\texttt{data } c \; (((x_i \; k_i))_i) \; \vec{alt}) \text{ locdecl } l \;\dashv\; \Delta'}$$

$$\Delta \;\not\ni\; l \text{ exp } n : T$$
$$\Delta \;\not\ni\; l \text{ loc } n : T$$
$$\Delta \; ; \;\vdash\; T \text{ type } l :: \texttt{typeK}$$
$$\overline{\Delta \;\vdash\; (\texttt{declare } n \; T) \text{ locdecl } l \;\dashv\; \Delta, l \text{ loc } n : T}$$

Fig. 11. Elaboration Judgments

$$\boxed{\Delta \;\vdash\; alt \text{ expalt } c \text{ on } \vec{ks} \text{ in } l \;\dashv\; \Delta'}$$

Exported constructor alternative alt for type constructor c with kind signatures $\vec{ks}$ elaborates nominal context Δ to nominal context Δ'

$$\Delta \;\not\ni\; l \text{ expcon } c \text{ as } [\vec{x'}](\vec{T'})c'$$
$$\Delta \;\not\ni\; l \text{ loccon } c \text{ as } [\vec{x'}](\vec{T'})c'$$
$$\frac{\forall j(\Delta \;;\; (x_i :: k_i)_i \;\vdash\; T_j :: \texttt{typeK})}{\begin{array}{c}\Delta \;\vdash\; (c\ \vec{T}) \text{ expalt } c' \text{ on } ((x_i\ k_i))_i \text{ in } l \\ \dashv\; \Delta, l \text{ expcon } c \text{ as } [\vec{x}](\vec{T})c'\end{array}}$$

$$\boxed{\Delta \;\vdash\; alt \text{ localt } c \text{ on } \vec{ks} \text{ in } l \;\dashv\; \Delta'}$$

Local constructor alternative alt for type constructor c with kind signatures $\vec{ks}$ elaborates nominal context Δ to nominal context Δ'

$$\Delta \;\not\ni\; l \text{ expcon } c \text{ as } [\vec{x'}](\vec{T'})c'$$
$$\Delta \;\not\ni\; l \text{ loccon } c \text{ as } [\vec{x'}](\vec{T'})c'$$
$$\frac{\forall j(\Delta \;;\; (x_i :: k_i)_i \;\vdash\; T_j :: \texttt{typeK})}{\begin{array}{c}\Delta \;\vdash\; (c\ \vec{T}) \text{ localt } c' \text{ on } ((x_i\ k_i))_i \text{ in } l \\ \dashv\; \Delta, l \text{ loccon } c \text{ as } [\vec{x}](\vec{T})c'\end{array}}$$

$$\boxed{\Delta \;\vdash\; D \text{ define } l \;\dashv\; \Delta'}$$

Definition D in module l elaborates nominal context Δ to nominal context Δ'

$$\frac{\begin{array}{c}\Delta \text{ permits } \texttt{l.n} : T \text{ in } l \\ \Delta \;;\; \vdash\; T \;\ni\; V \text{ term } l\end{array}}{\Delta \;\vdash\; (\texttt{define n } V) \text{ define } l \;\dashv\; \Delta, l \text{ def } n = V}$$

Fig. 12.   Elaboration Judgments (cont.)

$$\boxed{P \;\ggg\; \delta}$$

Program P generates environment δ

$$\frac{\forall i(M_i \;\gg\; \delta_i)}{(\texttt{program } \vec{M}) \;\ggg\; \vec{\delta}}$$

$$\boxed{M \;\gg\; \delta}$$

Module M generates environment δ

$$\frac{\forall i(\vec{n'} \;\ni\; n_i)}{\begin{array}{c}(\texttt{module } l\ \vec{l'} \text{ (exported } \vec{n'}) \overrightarrow{(c\ a)}\ ld\ \overrightarrow{(\texttt{define } n\ M)}) \\ \gg\; (n_i \mapsto M_i)_i\end{array}}$$

Fig. 13.   Declaration Environment Generation

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

Fig. 14.   Return Values of Plutus Core

$$\begin{array}{llll} \text{TCtx} & TK & ::= & \circ & \text{hol} \\ & & & (\texttt{arrT } TK\ T) & \text{left arrow contex} \\ & & & (\texttt{arrT } TV\ TK) & \text{right arrow contex} \\ & & & (\texttt{conT } qc\ TV^*\ TK\ T^*) & \text{type constructor contex} \\ & & & (\texttt{compT } TK) & \text{computation contex} \\ & & & (\texttt{forallT } x\ k\ TK) & \text{forall contex} \\ & & & (\texttt{appT } TK\ T) & \text{left app contex} \\ & & & (\texttt{appT } TV\ TK) & \text{right app contex} \end{array}$$

Fig. 15.   Grammar of Type Reduction Contexts

$$\begin{aligned} \circ\{T\} &= T \\ (\texttt{arrT } TK\ T')\{T\} &= (\texttt{arrT } TK\{T\}\ T') \\ (\texttt{arrT } TV\ TK)\{T\} &= (\texttt{arrT } TV\ TK\{T\}) \\ (\texttt{conT } qc\ \vec{TV}\ TK\ \vec{T'})\{T\} &= (\texttt{conT } qc\ \vec{TV}\ TK\{T\}\ \vec{T'}) \\ (\texttt{compT } TK)\{T\} &= (\texttt{compT } TK\{T\}) \\ (\texttt{forallT } x\ k\ TK)\{T\} &= (\texttt{forallT } x\ k\ TK\{T\}) \\ (\texttt{appT } TK\ T')\{T\} &= (\texttt{appT } TK\{T\}\ T') \\ (\texttt{appT } TV\ TK)\{T\} &= (\texttt{appT } TV\ TK\{T\}) \end{aligned}$$

Fig. 16.   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'

$$\frac{T \;\Rightarrow_{ty}\; T'}{TK\{T\} \;\rightarrow_{ty}\; TK\{T'\}}$$

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

Type T locally reduces to type T'

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

Fig. 17.   Type Reduction via Contextual Dynamics

$$\begin{array}{llll} \text{Ctx} & K & ::= & \circ & \text{hole} \\ & & & [K\ M] & \text{left app context} \\ & & & [V\ K] & \text{right app context} \\ & & & (\texttt{con } qc\ V^*\ K\ M^*) & \text{condata context} \\ & & & (\texttt{case } K\ C^*) & \text{case context} \\ & & & (\texttt{success } K) & \text{success context} \\ & & & (\texttt{bind } K\ x\ M) & \text{bind context} \\ & & & (\texttt{builtin } n\ V^*\ K\ M^*) & \text{builtin context} \end{array}$$

Fig. 18.   Grammar of Reduction Contexts

$$\circ\{N\} = N$$
$$[K\ M]\{N\} = [K\{N\}\ M]$$
$$[M\ K]\{N\} = [M\ K\{N\}]$$
$$(\texttt{con}\ qc\ \vec{V}\ K\ \vec{M})\{N\} = (\texttt{con}\ qc\ \vec{V}\ K\{N\}\ \vec{M})$$
$$(\texttt{case}\ K\ \vec{C})\{N\} = (\texttt{case}\ K\{N\}\ \vec{C})$$
$$(\texttt{success}\ K)\{N\} = (\texttt{success}\ K\{N\})$$
$$(\texttt{bind}\ K\ x\ M)\{N\} = (\texttt{bind}\ K\{N\}\ x\ M)$$

Fig. 19.   Context Insertion

$$\boxed{M\ \to_\delta^*\ R}$$

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

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

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

$$\frac{M\ \to_\delta\ \texttt{err}}{M\ \to_\delta^*\ \texttt{err}}$$

$$\boxed{M\ \to_\delta^n\ R}$$

Term M reduces in at most n steps to return value R in declaration environment δ

$$\frac{}{V\ \to_\delta^n\ (\texttt{ok}\ V)}$$

$$\frac{M\ \to_\delta\ (\texttt{ok}\ M')}{M\ \to_\delta^0\ \texttt{err}}$$

$$\frac{M\ \to_\delta\ (\texttt{ok}\ M')\qquad M'\ \to_\delta^n\ R}{M\ \to_\delta^{n+1}\ R}$$

$$\frac{M\ \to_\delta\ \texttt{err}}{M\ \to_\delta^n\ \texttt{err}}$$

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

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

$$\frac{M\ \Rightarrow_\delta\ (\texttt{ok}\ M')}{K\{M\}\ \to_\delta\ (\texttt{ok}\ K\{M'\})}$$

$$\frac{M\ \Rightarrow_\delta\ \texttt{err}}{K\{M\}\ \to_\delta\ \texttt{err}}$$

Fig. 20.   Reduction via Contextual Dynamics

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

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

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

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

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

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

Fig. 21.   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

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

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

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

Fig. 22.   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. 23.   Grammar of Instructions and Return Instructions

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

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

$$\frac{M \ \to^*_\delta \ \texttt{err}}{M \ \leadsto^*_{E,\delta} \ \texttt{err}}$$

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

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

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

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

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

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

$$\frac{\begin{array}{c} M \ \to^*_\delta \ (\texttt{ok } V) \\ V \ = \ (\texttt{bind } V_0 \ x \ M'_1) \\ V_0 \ \leadsto^*_{E,\delta} \ \texttt{err} \end{array}}{M \ \leadsto^*_{E,\delta} \ \texttt{err}}$$

$$\frac{\begin{array}{c} M \ \to^*_\delta \ (\texttt{ok } V) \\ V \ = \ (\texttt{bind } V_0 \ x \ M'_1) \\ V_0 \ \leadsto^*_{E,\delta} \ (\texttt{ok } V') \\ [V'/x]M'_1 \ \leadsto^*_{E,\delta} \ R \end{array}}{M \ \leadsto^*_{E,\delta} \ R}$$

Fig. 24.   Execution

$$\boxed{M \ \leadsto^n_{E,\delta} \ R}$$

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

$$\frac{M \ \to^n_\delta \ \texttt{err}}{M \ \leadsto^n_{E,\delta} \ \texttt{err}}$$

$$\frac{M \ \to^n_\delta \ (\texttt{ok } V) \qquad V \ \neq \ I}{M \ \leadsto^n_{E,\delta} \ \texttt{err}}$$

$$\frac{M \ \to^n_\delta \ (\texttt{ok } V) \qquad V \ = \ (\texttt{success } V')}{M \ \leadsto^n_{E,\delta} \ (\texttt{ok } V')}$$

$$\frac{M \ \to^n_\delta \ (\texttt{ok } V) \qquad V \ = \ (\texttt{failure})}{M \ \leadsto^n_{E,\delta} \ \texttt{err}}$$

$$\frac{M \ \to^n_\delta \ (\texttt{ok } V) \qquad V \ = \ (\texttt{txhash})}{M \ \leadsto^n_{E,\delta} \ (\texttt{ok } E_{txhash})}$$

$$\frac{M \ \to^n_\delta \ (\texttt{ok } V) \qquad V \ = \ (\texttt{blocknum})}{M \ \leadsto^n_{E,\delta} \ (\texttt{ok } E_{blocknum})}$$

$$\frac{M \ \to^n_\delta \ (\texttt{ok } V) \qquad V \ = \ (\texttt{blocktime})}{M \ \leadsto^n_{E,\delta} \ (\texttt{ok } E_{blocktime})}$$

$$\frac{\begin{array}{c} M \ \to^n_\delta \ (\texttt{ok } V) \\ V \ = \ (\texttt{bind } V_0 \ x \ M'_1) \\ V_0 \ \leadsto^{n'}_{E,\delta} \ \texttt{err} \end{array}}{M \ \leadsto^{n+n'}_{E,\delta} \ \texttt{err}}$$

$$\frac{\begin{array}{c} M \ \to^n_\delta \ (\texttt{ok } V) \\ V \ = \ (\texttt{bind } V_0 \ x \ M'_1) \\ V_0 \ \leadsto^{n'}_{E,\delta} \ (\texttt{ok } V') \\ [V'/x]M'_1 \ \leadsto^{n''}_{E,\delta} \ R \end{array}}{M \ \leadsto^{n+n'+n''+1}_{E,\delta} \ R}$$

Fig. 25.   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	$i_0 < i_1$
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	$f_0 < f_1$
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	$take(i, b)$
drop	i b	$drop(i, b)$
sha2_256	b	$sha2\_256(b)$
sha3_256	b	$sha3\_256(b)$
equalsByteString	b_0 b_1	$b_0 = b_1$

Fig. 26.   Builtin Reductions