Formal Specification of the Plutus Core Language (rev. 4)

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 3, 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 5, with auxiliary judgments in Figure 7. 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.

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 $\texttt{n} \mapsto M$. 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 9.

To give the computation rules for Plutus Core, we must define what the return values are of the language, as given in Figure 10. 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 $M \to_\delta^* R$ which means M eagerly reduces to R using declarations δ, in Figure 13. 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 $M \to_\delta^n R$, 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 $M \to_\delta R$, 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 $M \to_\delta^{max} R$.

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 $Integer$s, primitive floats as $Float$,

QualN	qn	::=	$l.n$	qualified name
QualC	qc	::=	$l.c$	qualified constructor
Tm	M	::=	x	variable
			qn	declared name
			(let M x M)	local declaration
			(lam x M)	λ abstraction
			[M M]	function application
			(con qc M^*)	constructed data
			(case M C^*)	case
			(success M)	success
			(failure)	failure
			(txhash)	transaction hash
			(blocknum)	block number
			(blocktime)	block time
			(bind M x M)	computation bind
			i	primitive integer
			f	primitive float
			b	primitive bytestring
			(builtin n M^*)	built-in function
			(isFun M)	function test
			(isCon M)	data test
			(isConName qc M)	named data test
			(isInteger M)	integer test
			(isFloat M)	float test
			(isByteString M)	bytestring test
Cl	C	::=	(cl qc (x^*) M)	case clause
Prg	G	::=	(program L^*)	program
Mod	L	::=	(module l id ed ld D^*)	module
ImpD	id	::=	(imported l^*)	import decls
ExpD	ed	::=	(exported ((c a)*) (n^*))	export decls
LocD	ld	::=	(local ((c a)*) (n^*))	local decls
Def	D	::=	(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

Fig. 1. Grammar of Plutus Core

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])$^+$ bytestring
Arity	a	::=	[0-9]$^+$ arity
Var	x	::=	[a-z][a-zA-Z0-9_′]* variable

Fig. 2. Lexical Grammar of Plutus Core

```
(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. 3. Example with Fibonacci and Map

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, however. $drop(x, y)$ 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

NomCtx	Δ	$::=$	ϵ	empty nominal context
			Δ, ν	non-empty nominal context
Nom	ν	$::=$	l loc n	local name
			l exp n	exported name
			l loccon $c : a$	local constructor
			l expcon $c : a$	exported constructor
VarCtx	Γ	$::=$	ϵ	empty variable context
			Γ, x	non-empty variable context
ModCtx	Λ	$::=$	ϵ	empty module context
			Λ, l	non-empty module context

Fig. 4. Contexts

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 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

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

Term M in module l is well-formed with names in nominal context Δ and variables in variable context Γ

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

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

$$\frac{\Delta \ ; \ \Gamma \ \vdash \ M_0 \text{ term } l \qquad \Gamma \ \not\ni \ x \qquad \Delta \ ; \ \Gamma, x \ \vdash \ M_1 \text{ term } l}{\Delta \ ; \ \Gamma \ \vdash \ \texttt{(let } M_0 \ x \ M_1\texttt{)} \text{ term } l}$$

$$\frac{\Delta \ ; \ \Gamma, x \ \vdash \ M \text{ term } l \qquad \Gamma \ \not\ni \ x}{\Delta \ ; \ \Gamma \ \vdash \ \texttt{(lam } x \ M\texttt{)} \text{ term } l}$$

$$\frac{\Delta \ ; \ \Gamma \ \vdash \ M_0 \text{ term } l \qquad \Delta \ ; \ \Gamma \ \vdash \ M_1 \text{ term } l}{\Delta \ ; \ \Gamma \ \vdash \ \texttt{[}M_0 \ M_1\texttt{]} \text{ term } l}$$

$$\frac{\Delta \text{ permits constructor } qc : |\vec{M}| \text{ in } l \qquad \forall i(\Delta \ ; \ \Gamma \ \vdash \ M_i \text{ term } l)}{\Delta \ ; \ \Gamma \ \vdash \ \texttt{(con } qc \ \vec{M}\texttt{)} \text{ term } l}$$

$$\frac{\Delta \ ; \ \Gamma \ \vdash \ M \text{ term } l \qquad \vec{C} \text{ has no repeated constructors} \qquad \forall i(\Delta \ ; \ \Gamma \ \vdash \ C_i \text{ clause } l)}{\Delta \ ; \ \Gamma \ \vdash \ \texttt{(case } M \ \vec{C}\texttt{)} \text{ term } l}$$

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

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

$$\frac{\Delta \ ; \ \Gamma \ \vdash \ M_0 \text{ term } l \qquad \Gamma \ \not\ni \ x \qquad \Delta \ ; \ \Gamma, x \ \vdash \ M_1 \text{ term } l}{\Delta \ ; \ \Gamma \ \vdash \ \texttt{(bind } M_0 \ x \ M_1\texttt{)} \text{ term } l}$$

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

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

$$\frac{}{\Delta \ ; \ \Gamma \ \vdash \ \texttt{(blocktime)} \text{ term } l}$$

Fig. 5. Scope Correctness

```
(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,

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

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

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

$$\frac{\forall i(\Delta\ ;\ \Gamma\ \vdash\ M_i\ \text{term}\ l)}{\Delta\ ;\ \Gamma\ \vdash\ (\texttt{builtin}\ n\ \vec{M})\ \text{term}\ l}$$

$$\frac{\Delta\ ;\ \Gamma\ \vdash\ M\ \text{term}\ l}{\Delta\ ;\ \Gamma\ \vdash\ (\texttt{isFun}\ M)\ \text{term}\ l}$$

$$\frac{\Delta\ ;\ \Gamma\ \vdash\ M\ \text{term}\ l}{\Delta\ ;\ \Gamma\ \vdash\ (\texttt{isCon}\ M)\ \text{term}\ l}$$

$$\frac{\Delta\ ;\ \Gamma\ \vdash\ M\ \text{term}\ l}{\Delta\ ;\ \Gamma\ \vdash\ (\texttt{isConName}\ n\ M)\ \text{term}\ l}$$

$$\frac{\Delta\ ;\ \Gamma\ \vdash\ M\ \text{term}\ l}{\Delta\ ;\ \Gamma\ \vdash\ (\texttt{isInteger}\ M)\ \text{term}\ l}$$

$$\frac{\Delta\ ;\ \Gamma\ \vdash\ M\ \text{term}\ l}{\Delta\ ;\ \Gamma\ \vdash\ (\texttt{isFloat}\ M)\ \text{term}\ l}$$

$$\frac{\Delta\ ;\ \Gamma\ \vdash\ M\ \text{term}\ l}{\Delta\ ;\ \Gamma\ \vdash\ (\texttt{isByteString}\ M)\ \text{term}\ l}$$

Fig. 6. Scope Correctness (cont.)

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.

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

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

$$\frac{\Delta\ \ni\ l'\ \text{exp}\ n}{\Delta\ \text{permits}\ l'.n\ \text{in}\ l}$$

$$\frac{\Delta\ \ni\ l'\ \text{loc}\ n \qquad l = l'}{\Delta\ \text{permits}\ l'.n\ \text{in}\ l}$$

$\boxed{\Delta\ \text{permits constructor}\ qc : a\ \text{in}\ l}$

Nominal context Δ permits the use of qualified constructor name qc with arity a in module l

$$\frac{\Delta\ \ni\ l'\ \text{expcon}\ c : a}{\Delta\ \text{permits constructor}\ l'.c : a\ \text{in}\ l}$$

$$\frac{\Delta\ \ni\ l'\ \text{loccon}\ c : a \qquad l = l'}{\Delta\ \text{permits constructor}\ l'.c : a\ \text{in}\ l}$$

$\boxed{\Delta\ ;\ \Gamma\ \vdash\ C\ \text{clause}\ l}$

Clause C in module l is well-formed with names in nominal context Δ and variables in variable context Γ

$$\frac{\Delta\ \text{permits constructor}\ qc : |\vec{x}|\ \text{in}\ l \qquad \forall i(\Gamma\ \not\ni\ x_i) \qquad \Delta\ ;\ \Gamma, \vec{x}\ \vdash\ M\ \text{term}\ l}{\Delta\ ;\ \Gamma\ \vdash\ (\texttt{cl}\ qc\ (\tilde{x})\ M)\ \text{clause}\ l}$$

Fig. 7. Auxiliary Scope Correctness Judgments

$\boxed{\Delta \ \vdash \ ed \text{ edecl } l \ \dashv \ \Delta'}$

Exported declarations ed in module l are well-formed nominal context Δ, producing new nominal context Δ'

$$\frac{\begin{array}{cc}\forall i(\Delta \not\ni l \text{ exp } n_i) & \forall j(\Delta \not\ni l \text{ expcon } c_j : a') \\ \forall i(\Delta \not\ni l \text{ loc } n_i) & \forall j(\Delta \not\ni l \text{ loccon } c_j : a'') \\ \vec{n} \text{ all distinct} & \vec{c} \text{ all distinct}\end{array}}{\begin{array}{l}\Delta \ \vdash \ \texttt{(exported (}\vec{n}\texttt{) (}\overrightarrow{\texttt{(}c\ a\texttt{)}}\texttt{))} \text{ edecl } l \\ \quad \dashv \ \Delta, (l \text{ exp } n_i)_i, (l \text{ expcon } c_j : a_j)_j\end{array}}$$

$\boxed{\Delta \ \vdash \ ld \text{ ldecl } l \ \dashv \ \Delta'}$

Local declarations ld in module l are well-formed nominal context Δ, producing new nominal context Δ'

$$\frac{\begin{array}{cc}\forall i(\Delta \not\ni l \text{ exp } n_i) & \forall j(\Delta \not\ni l \text{ expcon } c_j : a') \\ \forall i(\Delta \not\ni l \text{ loc } n_i) & \forall j(\Delta \not\ni l \text{ loccon } c_j : a'') \\ \vec{n} \text{ all distinct} & \vec{c} \text{ all distinct}\end{array}}{\begin{array}{l}\Delta \ \vdash \ \texttt{(local (}\vec{n}\texttt{) (}\overrightarrow{\texttt{(}c\ a\texttt{)}}\texttt{))} \text{ ldecl } l \\ \quad \dashv \ \Delta, (l \text{ loc } n_i)_i, (l \text{ loccon } c_j : a_j)_j\end{array}}$$

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

Module L is well-formed with names in nominal context Δ, producing new nominal context Δ'

$$\frac{\begin{array}{c}\Delta \not\ni l \text{ mod} \\ \Delta_0 = \Delta \text{ restricted to } l, \vec{l'} \\ \Delta_0 \ \vdash \ ed \text{ edecl } l \ \dashv \ \Delta_1 \\ \Delta_1 \ \vdash \ ld \text{ ldecl } l \ \dashv \ \Delta_2 \\ \Delta_2 \ \vdash \ D_i \text{ defs } l \\ \Delta_2 \text{ matches } \vec{D}\end{array}}{\begin{array}{l}\Delta \ \vdash \ \texttt{(module } l \texttt{ (imported } \vec{l'}\texttt{))} \ ed \ ld \ \vec{D}\texttt{)} \text{ module} \\ \quad \dashv \Delta \cup \Delta_2, l \text{ mod}\end{array}}$$

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

Program G is well-formed in nominal context Δ, producing new nominal context Δ'

$$\frac{\Delta_0 = \varnothing \qquad \forall i(\Delta_i \ \vdash \ L_i \text{ module } \dashv \Delta_{i+1})}{\texttt{(program } \vec{L}\texttt{)} \text{ program } \dashv \Delta_n}$$

Fig. 8. Auxiliary Scope Correctness Judgments (cont.)

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

Program P generates environment δ

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

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

Module M generates environment δ

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

Fig. 9. Declaration Environment Generation

Ret	R	::=	$\texttt{(ok } M\texttt{)}$	returned value
			$\texttt{err}$	error

Fig. 10. Return Values of Plutus Core

Ctx	K	::=	$\circ$	hole
			$\texttt{(let } K\ x\ M\texttt{)}$	let context
			$\texttt{[}K\ M\texttt{]}$	left app context
			$\texttt{[}V\ K\texttt{]}$	right app context
			$\texttt{(con } qc\ V^*\ K\ M^*\texttt{)}$	condata context
			$\texttt{(case } K\ C^*\texttt{)}$	case context
			$\texttt{(success } K\texttt{)}$	success context
			$\texttt{(bind } K\ x\ M\texttt{)}$	bind context
			$\texttt{(builtin } n\ V^*\ K\ M^*\texttt{)}$	builtin context
			$\texttt{(isFun } K\texttt{)}$	function test context
			$\texttt{(isCon } K\texttt{)}$	data test context
			$\texttt{(isConName } qn\ K\texttt{)}$	named data text context
			$\texttt{(isInteger } K\texttt{)}$	int test context
			$\texttt{(isFloat } K\texttt{)}$	float test context
			$\texttt{(isByteString } K\texttt{)}$	bytestring test context

Fig. 11. Grammar of Reduction Contexts

$$
\begin{aligned}
\circ\{N\} &= N \\
(\texttt{let}\ K\ x\ M)\{N\} &= (\texttt{let}\ K\{N\}\ x\ M) \\
[K\ M]\{N\} &= [K\{N\}\ M] \\
[M\ K]\{N\} &= [M\ K\{N\}] \\
(\texttt{con}\ qc\ \vec{V}\ K\ \vec{M})\{N\} &= (\texttt{con}\ qc\ \vec{V}\ K\{N\}\ \vec{M}) \\
(\texttt{case}\ K\ \vec{C})\{N\} &= (\texttt{case}\ K\{N\}\ \vec{C}) \\
(\texttt{success}\ K)\{N\} &= (\texttt{success}\ K\{N\}) \\
(\texttt{bind}\ K\ x\ M)\{N\} &= (\texttt{bind}\ K\{N\}\ x\ M) \\
(\texttt{isFun}\ K)\{N\} &= (\texttt{isFun}\ K\{N\}) \\
(\texttt{isCon}\ K)\{N\} &= (\texttt{isCon}\ K\{N\}) \\
(\texttt{isConName}\ qc\ K)\{N\} &= (\texttt{isConName}\ qc\ K\{N\}) \\
(\texttt{isInteger}\ K)\{N\} &= (\texttt{isInteger}\ K\{N\}) \\
(\texttt{isFloat}\ K)\{N\} &= (\texttt{isFloat}\ K\{N\}) \\
(\texttt{isByteString}\ K)\{N\} &= (\texttt{isByteString}\ K\{N\})
\end{aligned}
$$

Fig. 12. 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. 13. Reduction via Contextual Dynamics

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

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

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

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

$$\frac{V\ \neq\ (\texttt{lam}\ x\ M')}{[V\ V']\ \Rightarrow_\delta\ \texttt{err}}$$

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

$$\frac{V\ \neq\ (\texttt{con}\ qc\ \vec{V})}{(\texttt{case}\ V\ \vec{C})\ \Rightarrow_\delta\ \texttt{err}}$$

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

Fig. 14. Local Reduction

$$\frac{V\ =\ (\texttt{lam}\ x\ M)}{(\texttt{isFun}\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.True}))}$$

$$\frac{V\ \neq\ (\texttt{lam}\ x\ M)}{(\texttt{isFun}\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.False}))}$$

$$\frac{V\ =\ (\texttt{con}\ qc\ \vec{V'})}{(\texttt{isCon}\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.True}))}$$

$$\frac{V\ \neq\ (\texttt{con}\ qc\ \vec{V'})}{(\texttt{isCon}\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.False}))}$$

$$\frac{V\ =\ (\texttt{con}\ qc\ \vec{V'})}{(\texttt{isConName}\ qc\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.True}))}$$

$$\frac{V\ \neq\ (\texttt{con}\ qc\ \vec{V'})}{(\texttt{isConName}\ qc\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.False}))}$$

$$\frac{V\ =\ i}{(\texttt{isInteger}\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.True}))}$$

$$\frac{V\ \neq\ i}{(\texttt{isInteger}\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.False}))}$$

$$\frac{V\ =\ f}{(\texttt{isFloat}\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.True}))}$$

$$\frac{V\ \neq\ f}{(\texttt{isFloat}\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.False}))}$$

$$\frac{V\ =\ b}{(\texttt{isByteString}\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.True}))}$$

$$\frac{V\ \neq\ b}{(\texttt{isByteString}\ V)\ \Rightarrow_\delta\ (\texttt{ok}\ (\texttt{con Prelude.False}))}$$

Fig. 15. Local Reduction (cont.)

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

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

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

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

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

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

Fig. 16. Case Matching

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

Fig. 17. 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

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

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

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

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

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

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

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

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

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

Fig. 18. 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

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

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

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

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

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

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

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

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

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

Fig. 19. 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. 20. Builtin Reductions