# Formal Specification of the Plutus Core Language (rev. 10)

## I. Plutus Core

Plutus Core is a typed, strict, eagerly-reduced $\lambda$-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 and 2, using a modified s-expression format. As is standard in $\lambda$-calculi, we have variables, $\lambda$-abstractions, and application. In addition to this, there are also polymorphic 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 ($[T\ T^+]$) and in terms ($[M\ M^+]$). This is to be understood as a convenient form of syntactic sugar for iterated binary application associated to the left, and the formal rules treat only the binary case.

As an example, consider the program in Figure 3, 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 4, we define the grammar of the 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, exported types, exported terms, and then term names and their definitions, term constructors, type constructors, and type names and their definitions.

We refer to the components of a nominal context by dotted names ($\Theta.\bar{l}$, $\Theta.\overline{\hat{\kappa}|\hat{\nu}}$, $\Theta.\overline{\hat{c}|\hat{n}}$, and $\Theta.\overline{nj}$). Variable contexts contain information about the nature of variables — type variables with their kind, and term variables with their type. The overall context $\Theta$ consists of nominal and variable contexts, along with the name of the current module being elaborated and the name of imported modules. As with nominal contexts, we also refer to these by dotted names ($\Theta.l$, $\Theta.\bar{l}$, $\Theta.\Delta$, and $\Theta.\Gamma$) In the inference rules, we use $\Theta, \alpha :: K$ to mean $\Theta$ with it's variable context $\Gamma$ extended with $\alpha :: K$, and $\Theta, x : A$ to mean $\Theta$ with it's variable context $\Gamma$ extended with $x : A$.

Then, in Figure 5, we define what it means for a type construct 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 6, we define the type checking judgment that explains when a type contains a term. This is defined together with Figure 7's type synthesis judgment, which explains how a term synthesizes a type. Together, these two judgments constitute a standard bidirectional type theory [1] [2].

A number of auxiliary judgments are defined in Figure 8. In particular, we define when a type contains a clause for that type's constructors, and how that then synthesizes a type. We also define what it means for a qualified name to be permitted for use. Finally, we define the various elaboration judgments in Figure 9, which explain how declarations, modules, and programs elaborate out to complete nominal contexts. Declarations for the types and definitions of terms are separated into two distinct forms, rather than a single construct. One reason for this is that it makes type checking mutual recursion relatively simple, because all the types of names can be put before use sites.

Finally, type synthesis for built-in operations ($n$ is $(\texttt{fun}\ \overline{S}\ T)$) is given in tabular form rather than in inference rule form, in Figure 22, which also gives the reduction semantics. The types in the arguments column constitute $\overline{T}$, and the type in the return column constitutes $T'$, in the judgment form. The same is done for synthesis of built-in computations ($n$ is $T$).

## 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 $\delta$, which we will represent by a list of items of the form $\hat{n} \mapsto M$. 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.

To give the computation rules for Plutus Core, we must define what the return values are of the language, as given in Figure 18. 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 $\delta$, in Figure 15. 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

| | | | | |
|---|---|---|---|---|
| Term | $M, N$ | ::= | $x$ | variable |
| | | | $\hat{n}$ | declared name |
| | | | (isa $T$ $M$) | type annotation |
| | | | (abs $\alpha$ $M$) | type abstraction |
| | | | (inst $M$ $T$) | type instantiation |
| | | | (lam $x$ $M$) | $\lambda$ abstraction |
| | | | [$M$ $M^+$] | function application |
| | | | (con $\hat{c}$ $M^*$) | constructed data |
| | | | (case $M$ $C^*$) | case |
| | | | (success $M$) | success |
| | | | (failure) | failure |
| | | | (compbuiltin $cbi$) | computation builtin |
| | | | (bind $M$ $x$ $M$) | computation bind |
| | | | $i$ | primitive integer |
| | | | $f$ | primitive float |
| | | | $b$ | primitive bytestring |
| | | | (builtin $bi$ $M^*$) | built-in function |
| Clause | $C$ | ::= | ($\hat{c}$ ($x^*$) $M$) | case clause |
| Program | $G$ | ::= | (program $L^*$) | program |
| Module | $L$ | ::= | (module $l$ $id$ $ed$ $D^*$) | module |
| Imported | $id$ | ::= | (import $l^*$) | import decls |
| Exported | $ed$ | ::= | (export ($tx^*$) ($mx^*$)) | export decls |
| Declaration | $D$ | ::= | $dd$ | data decl. |
| | | | $td$ | type decl. |
| | | | $df$ | term defn. |
| Type | $R, S, T$ | ::= | $\alpha$ | type variable |
| | | | $\hat{\nu}$ | declared type name |
| | | | (fun $T$ $T$) | function type |
| | | | (con $\hat{\kappa}$ $T^*$) | type constructor |
| | | | (comp $T$) | computation type |
| | | | (forall $\alpha$ $K$ $T$) | polymorphic type |
| | | | (bytestring) | bytestring |
| | | | (integer) | integer |
| | | | (float) | float |
| | | | (lam $\alpha$ $K$ $T$) | type abstraction |
| | | | [$T$ $T^+$] | type application |
| Kind | $J, K$ | ::= | (type) | type kind |
| | | | (fun $K$ $K$) | arrow kind |
| Type Export | $tx$ | ::= | $\nu$ | type export |
| | | | ($\kappa$ ($c^*$)) | data export |
| Term Export | $mx$ | ::= | $n$ | term export |
| Data Declaration | $dd$ | ::= | (data $\kappa$ ($ks^*$) $alt^*$) | data decl. |
| Type Declaration | $td$ | ::= | (type $\nu$ $T$) | type decl. |
| Kind Signature | $ks$ | ::= | ($\alpha$ $K$) | kind signature |
| Alternative | $alt$ | ::= | ($c$ $T^*$) | alternative |
| Term Definition | $df$ | ::= | (define $n$ $M$) | name definition |

Fig. 1. Grammar of Plutus Core

$R$ using $\delta$ 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. $\beta$) 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

| | | | | |
|---|---|---|---|---|
| Var | $x$ | ::= | [a-z][a-zA-Z0-9_′]* | term variable |
| TyVar | $\alpha$ | ::= | [a-z][a-zA-Z0-9_′]* | type variable |
| QualTmN | $\hat{n}$ | ::= | $l.n$ | qualified term name |
| QualTyN | $\hat{\nu}$ | ::= | $l.\nu$ | qualified type name |
| QualTmC | $\hat{c}$ | ::= | $l.c$ | qualified term constructor |
| QualTyC | $\hat{\kappa}$ | ::= | $l.\kappa$ | qualified type constructor |
| TmName | $n$ | ::= | [a-z][a-zA-Z0-9_′]* | term name |
| TyName | $\nu$ | ::= | [a-z][a-zA-Z0-9_′]* | type name |
| Mod | $l$ | ::= | [A-Z][a-zA-Z0-9_′]* | module name |
| Con | $c$ | ::= | [A-Z][a-zA-Z0-9_′]* | term constructor name |
| TyCon | $\kappa$ | ::= | [A-Z][a-zA-Z0-9_′]* | type constructor name |
| CompBuiltin | $cbi$ | ::= | [a-z][a-zA-Z0-9_′]* | comp. builtin 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 |
| Builtin | $bi$ | ::= | [a-z][a-zA-Z0-9_′]* | builtin name |

Fig. 2. Lexical Grammar of Plutus Core

*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, 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 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)`, `(compbuiltin` $n$`)`, 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 $\delta$, 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.

```
(program
  (module Ex
    (import Prelude)
    (export ((Nat (Zero Suc)) (List (Nil Cons))) (fibonacci map))
    (data Nat () (Zero) (Suc (con Prelude.Nat)))
    (data List ((a (type))) (Nil) (Cons a (con Prelude.List a)))
    (declare fibonacci (fun (con Prelude.Nat) (con Prelude.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 Prelude.List a) (con Prelude.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. 3. Example with Fibonacci and Map

| | | | | |
|---|---|---|---|---|
| NomCtx | $\Delta$ | ::= | $L^*; D^*$ | nominal context |
| VarCtx | $\Gamma$ | ::= | $j^*$ | variable context |
| VarJ | $j$ | ::= | $\alpha :: K$ | type variable |
| | | | $x : T$ | term variable |
| Ctx | $\Theta$ | ::= | $l; l^*; \Delta; \Gamma$ | compound context |

Fig. 4. Contexts

## REFERENCES


[1] Pfenning, F. *Lecture Notes on Bidirectional Type Checking*. 2004. https://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf

[2] Christiansen, D. R. *Bidirectional Typing Rules: A Tutorial*. http://www.davidchristiansen.dk/tutorials/bidirectional.pdf

$\boxed{\Theta \vdash T :: K}$

In context $\Theta$, type $T$ has kind $K$

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

$$\frac{\Theta \vdash \hat{\nu} :: K}{\Theta \vdash \hat{\nu} :: K}\ \text{tyname}$$

$$\frac{\Theta \vdash S :: \texttt{(type)} \qquad \Theta \vdash T :: \texttt{(type)}}{\Theta \vdash \texttt{(fun } S\ T\texttt{)} :: \texttt{(type)}}\ \text{fun}$$

$$\frac{\Theta \vdash \hat{\kappa} :: \texttt{(fun } \overline{K}\ \texttt{(type))} \qquad |\overline{K}| = |\overline{T}| \qquad \Theta \vdash \overline{T} :: \overline{K}}{\Theta \vdash \texttt{(con } \hat{\kappa}\ \overline{T}\texttt{)} :: \texttt{(type)}}\ \text{tycon}$$

$$\frac{\Theta \vdash T :: \texttt{(type)}}{\Theta \vdash \texttt{(comp } T\texttt{)} :: \texttt{(type)}}\ \text{comp}$$

$$\frac{\Theta, \alpha :: K \vdash T :: \texttt{(type)}}{\Theta \vdash \texttt{(forall } \alpha\ K\ T\texttt{)} :: \texttt{(type)}}\ \text{forall}$$

$$\frac{}{\Theta \vdash \texttt{(integer)} :: \texttt{(type)}}\ \text{integer}$$

$$\frac{}{\Theta \vdash \texttt{(float)} :: \texttt{(type)}}\ \text{float}$$

$$\frac{}{\Theta \vdash \texttt{(bytestring)} :: \texttt{(type)}}\ \text{bytestring}$$

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

$$\frac{\Theta \vdash S :: \texttt{(fun } J\ K\texttt{)} \qquad \Theta \vdash T :: J}{\Theta \vdash \texttt{[}S\ T\texttt{]} :: K}\ \text{tyapp}$$

Fig. 5. Type Well-formedness

$\boxed{\Theta \vdash T \ni M}$

In context $\Theta$, type $T$ checks term $M$

$$\frac{\Theta, \alpha :: K \vdash T \ni M}{\Theta \vdash \texttt{(forall } \alpha\ k\ T\texttt{)} \ni \texttt{(abs } \alpha\ M\texttt{)}}\ \text{abs}$$

$$\frac{\Theta, x : S \vdash T \ni M}{\Theta \vdash \texttt{(fun } S\ T\texttt{)} \ni \texttt{(lam } x\ M\texttt{)}}\ \text{lam}$$

$$\frac{\Theta \vdash \hat{c} : \texttt{(forall } \overline{\alpha}\ \overline{K}\ \texttt{(fun } \overline{S}\ \texttt{[}\hat{\kappa}\ \overline{\alpha}\texttt{]))} \qquad \Theta \vdash [\![[\overline{T}/\overline{\alpha}]S_i]\!] \ni M_i}{\Theta \vdash \texttt{(con } \hat{\kappa}\ \overline{T}\texttt{)} \ni \texttt{(con } \hat{c}\ \overline{M}\texttt{)}}\ \text{con}$$

$\overline{C}$ has no repeated constructors and covers all $S'$ constructors

$$\frac{\Theta \vdash M \in S \qquad \Theta; [\![S]\!] \vdash T \ni \overline{C}}{\Theta \vdash T \ni \texttt{(case } M\ \overline{C}\texttt{)}}\ \text{case}$$

$$\frac{\Theta \vdash T \ni M}{\Theta \vdash \texttt{(comp } T\texttt{)} \ni \texttt{(success } M\texttt{)}}\ \text{success}$$

$$\frac{}{\Theta \vdash \texttt{(comp } T\texttt{)} \ni \texttt{(failure)}}\ \text{failure}$$

$$\frac{\Theta \vdash M \in S \qquad T = S}{\Theta \vdash T \ni M}\ \text{dir-change}$$

Fig. 6. Type Checking

$\boxed{\Theta \vdash M \in T}$

In context $\Theta$, term $M$ synthesizes type $T$

$$\frac{\Theta.\Gamma \ni x : T}{\Theta \vdash x \in T}\ \text{var}$$

$$\frac{\Theta \vdash \hat{n} : T}{\Theta \vdash \hat{n} \in T}\ \text{name}$$

$$\frac{\Theta \vdash T \ni M}{\Theta \vdash \texttt{(isa}\ T\ M\texttt{)} \in T}\ \text{isa}$$

$$\frac{\Theta \vdash M \in \texttt{(forall}\ \alpha\ K\ T\texttt{)} \quad \Theta \vdash S :: K}{\Theta \vdash \texttt{(inst}\ M\ S\texttt{)} \in [S/\alpha]T}\ \text{inst}$$

$$\frac{\Theta \vdash M \in \texttt{(fun}\ S\ T\texttt{)} \quad \Theta \vdash S \ni N}{\Theta \vdash \texttt{[}M\ N\texttt{]} \in T}\ \text{app}$$

$$\frac{cbi \text{ is } T}{\Theta \vdash \texttt{(compbuiltin}\ cbi\texttt{)} \in \texttt{(comp}\ T\texttt{)}}\ \text{compbuiltin}$$

$$\frac{\Theta \vdash M \in \texttt{(comp}\ S\texttt{)} \quad \Theta, x : S \vdash N \in \texttt{(comp}\ T\texttt{)}}{\Theta \vdash \texttt{(bind}\ M\ x\ N\texttt{)} \in \texttt{(comp}\ T\texttt{)}}\ \text{bind}$$

$$\frac{}{\Theta \vdash i \in \texttt{(integer)}}\ \text{intval}$$

$$\frac{}{\Theta \vdash f \in \texttt{(float)}}\ \text{floatval}$$

$$\frac{}{\Theta \vdash b \in \texttt{(bytestring)}}\ \text{bytestringval}$$

$$\frac{bi \text{ is } \texttt{(fun}\ \overline{S}\ T\texttt{)} \quad |\overline{S}| = |\overline{M}| \quad \Theta \vdash \overline{S} \ni \overline{M}}{\Theta \vdash \texttt{(builtin}\ bi\ \overline{M}\texttt{)} \in T}\ \text{builtin}$$

Fig. 7. Type Synthesis

$\boxed{\Theta; S \vdash T \ni C}$

In context $\Theta$ under type $S$, clause $C$ has type $T$

$$\frac{\Theta \vdash \hat{c} : \texttt{(forall } \overline{\alpha}\ \overline{K}\ \texttt{ (fun } \overline{R}\ \texttt{ [}\hat{\kappa}\ \overline{\alpha}\texttt{] ))} \qquad |\overline{x}| = |\overline{R}| \qquad \Theta, \overline{x} : [\overline{S}/\overline{\alpha}]\overline{R} \vdash T \ni M}{\Theta; \texttt{(con } \hat{\kappa}\ \overline{S}\texttt{)} \vdash T \ni \texttt{(}\hat{c}\ \texttt{(}\overline{x}\texttt{)}\ M\texttt{)}}\ \text{clause}$$

$\boxed{\Theta \vdash \hat{\nu} :: K}$

In context $\Theta$, qualified type name $\hat{\nu}$ has kind $K$

$$\frac{\Theta.l = l \qquad \Theta.\Delta.\overline{D} \ni \texttt{(type } \nu\ T\texttt{)} \qquad \Theta \vdash T :: K}{\Theta \vdash l.\nu :: K}\ \text{tyname-valid-local}$$

$$\frac{\begin{array}{c}\Theta.\bar{l} \ni l \\ \Theta.\Delta.\overline{L} \ni \texttt{(module } l\ id\ \texttt{ (export (}\overline{tx}\texttt{) (}\overline{mx}\texttt{)) } \overline{D}\texttt{)} \\ \overline{tx} \ni \nu \qquad \overline{D} \ni \texttt{(type } \nu\ T\texttt{)} \qquad \Theta \vdash T :: K\end{array}}{\Theta \vdash l.\nu :: K}\ \text{tyname-valid-import}$$

$\boxed{\Theta \vdash \hat{\kappa} :: \texttt{(fun } \overline{K}\ \texttt{ (type))}}$

In context $\Theta$, qualified type constructor name $\hat{\kappa}$ has kind $\texttt{(fun } \overline{K}\ \texttt{ (type))}$

$$\frac{\Theta.l = l \qquad \Theta.\Delta.\overline{D} \ni \texttt{(data } \kappa\ \texttt{ ((}\overline{\alpha}\ \overline{K}\texttt{)) } \overline{alt}\texttt{)}}{\Theta \vdash l.\kappa :: \texttt{(fun } \overline{K}\ \texttt{ (type))}}\ \text{tycon-valid-local}$$

$$\frac{\begin{array}{c}\Theta.\bar{l} \ni l \\ \Theta.\Delta.\overline{L} \ni \texttt{(module } l\ id\ \texttt{ (export (}\overline{tx}\texttt{) (}\overline{mx}\texttt{)) } \overline{D}\texttt{)} \\ \overline{tx} \ni \texttt{(}\kappa\ \texttt{(}\overline{c}\texttt{))} \qquad \overline{D} \ni \texttt{(data } \kappa\ \texttt{ ((}\overline{\alpha}\ \overline{K}\texttt{)) } \overline{alt}\texttt{)}\end{array}}{\Theta \vdash l.\kappa :: \texttt{(fun } \overline{K}\ \texttt{ (type))}}\ \text{tycon-valid-import}$$

$\boxed{\Theta \vdash \hat{n} : T}$

In context $\Theta$, qualified name $\hat{n}$ has type $T$

$$\frac{\Theta.l = l \qquad \Theta.\Delta.\overline{D} \ni \texttt{(declare } n\ T\texttt{)}}{\Theta \vdash l.n : T}\ \text{name-valid-local}$$

$$\frac{\begin{array}{c}\Theta.\bar{l} \ni l \\ \Theta.\Delta.\overline{L} \ni \texttt{(module } l\ id\ \texttt{ (export (}\overline{tx}\texttt{) (}\overline{mx}\texttt{)) } \overline{D}\texttt{)} \\ \overline{mx} \ni n \qquad \overline{D} \ni \texttt{(declare } n\ T\texttt{)}\end{array}}{\Theta \vdash l.n : T}\ \text{name-valid-import}$$

$\boxed{\Theta \vdash \hat{c} : \texttt{(forall } \overline{\alpha}\ \overline{K}\ \texttt{ (fun } \overline{T}\ \texttt{ [}\hat{\kappa}\ \overline{\alpha}\texttt{] ))}}$

In context $\Theta$, qualified constructor name $\hat{c}$ has type parameters $\overline{\alpha}$ of kinds $\overline{K}$, argument types $\overline{T}$, and return type constructor $\hat{\kappa}$

$$\frac{\begin{array}{c}\Theta.l = l \\ \Theta.\Delta.\overline{D} \ni \texttt{(data } \kappa\ \texttt{ ((}\overline{\alpha}\ \overline{K}\texttt{)) } \overline{alt}\texttt{)} \\ \overline{alt} \ni \texttt{(}c\ \overline{T}\texttt{)}\end{array}}{\Theta \vdash l.c : \texttt{(forall } \overline{\alpha}\ \overline{K}\ \texttt{ (fun } \overline{T}\ \texttt{ [}l.\kappa\ \overline{\alpha}\texttt{] ))}}\ \text{con-valid-local}$$

$$\frac{\begin{array}{c}\Theta.\bar{l} \ni l \\ \Theta.\Delta.\overline{L} \ni \texttt{(module } l\ id\ \texttt{ (export (}\overline{tx}\texttt{) (}\overline{mx}\texttt{)) } \overline{D}\texttt{)} \\ \overline{tx} \ni \texttt{(}\kappa\ \texttt{(}\overline{c}\texttt{))} \qquad \overline{c} \ni c \\ \overline{D} \ni \texttt{(data } \kappa\ \texttt{ ((}\overline{\alpha}\ \overline{K}\texttt{)) } \overline{alt}\texttt{)} \qquad \overline{alt} \ni \texttt{(}c\ \overline{T}\texttt{)}\end{array}}{\Theta \vdash l.c : \texttt{(forall } \overline{\alpha}\ \overline{K}\ \texttt{ (fun } \overline{T}\ \texttt{ [}l.\kappa\ \overline{\alpha}\texttt{] ))}}\ \text{con-valid-import}$$

Fig. 8. Auxiliary Judgments

$\boxed{\vdash G}$

Program $G$ is valid

$$\frac{\begin{array}{c}\epsilon \vdash L_0 \\ L_0 \vdash L_1 \\ L_0, L_1 \vdash L_2 \\ \vdots \end{array}}{\vdash \texttt{(program } \overline{L}\texttt{)}} \text{ program-valid}$$

$\boxed{\overline{L} \vdash L}$

Module $L$ is valid in the context of modules $\overline{L}$

$$\frac{\begin{array}{c}\overline{L} \not\ni \texttt{(module } l\ id'\ ed'\ \overline{D}'\texttt{)} \\ \Delta_{\bar{l}} \ni \bar{l} \\ l; \bar{l}; \Delta_{\overline{L}}; \epsilon \vdash D_0 \\ l; \bar{l}; \Delta_{\overline{L}}; D_0 \vdash D_1 \\ l; \bar{l}; \Delta_{\overline{L}}; D_0, D_1 \vdash D_2 \\ \vdots \end{array}}{\overline{L} \vdash \texttt{(module } l\ \texttt{(import } \bar{l}\texttt{)}\ ed\ \overline{D}\texttt{)}} \text{ module-valid}$$

$\boxed{l; \bar{l}; \Delta \vdash D}$

Declaration $D$ in module $l$, with imported modules $\bar{l}$, is valid in nominal context $\Delta$

$$\frac{\Delta \not\ni l.\kappa \qquad l; \bar{l}; \Delta \vdash \overline{alt} \text{ alt } c \text{ on } \texttt{(}\overline{\alpha}\ \overline{K}\texttt{)}}{l; \bar{l}; \Delta \vdash \texttt{(data } \kappa\ \texttt{(}\texttt{(}\overline{\alpha}\ \overline{K}\texttt{)}\texttt{)}\ \overline{alt}\texttt{)}} \text{ decl-valid-data}$$

$$\frac{\Delta \not\ni l.\nu \qquad l; \bar{l}; \Delta; \epsilon \vdash T :: K \qquad T \text{ tyval}}{l; \bar{l}; \Delta \vdash \texttt{(type } \nu\ T\texttt{)}} \text{ decl-valid-type}$$

$$\frac{\begin{array}{c}\Delta \not\ni l.n = M' \\ M \text{ val} \qquad l; \bar{l}; \Delta; \epsilon \vdash M \in T\end{array}}{l; \bar{l}; \Delta \vdash \texttt{(define } n\ M\texttt{)}} \text{ decl-valid-define}$$

$\boxed{l; \bar{l}; \Delta \vdash alt \text{ alt } \kappa \text{ on } \overline{ks}}$

Constructor alternative $alt$ for type constructor $\kappa$ with kind signatures $\overline{ks}$ in module $l$ importing $\bar{l}$ is valid innominal context $\Delta$

$$\frac{\Delta \not\ni l.c \qquad l; \bar{l}; \Delta; \overline{\alpha} :: \overline{K} \vdash \overline{T} :: \texttt{(type)}}{l; \bar{l}; \Delta \vdash \texttt{(}c\ \overline{T}\texttt{)} \text{ alt } \kappa \text{ on } \texttt{(}\overline{\alpha}\ \overline{K}\texttt{)}} \text{ alt-valid}$$

Fig. 9. Elaboration Judgments

| | | | | |
|---|---|---|---|---|
| TCtx | $H$ | ::= | $\circ$ | hole |
| | | | $\texttt{(fun } H\ T\texttt{)}$ | left arrow |
| | | | $\texttt{(fun } U\ H\texttt{)}$ | right arrow |
| | | | $\texttt{(con } \hat{\kappa}\ U^*\ H\ T^*\texttt{)}$ | type constructor |
| | | | $\texttt{(comp } H\texttt{)}$ | computation |
| | | | $\texttt{(forall } \alpha\ K\ H\texttt{)}$ | forall |
| | | | $[H\ T]$ | left app |
| | | | $[U\ H]$ | right app |

Fig. 10. Grammar of Type Reduction Contexts

$$
\begin{aligned}
\circ\{T\} &= T \\
(\texttt{fun}\ H\ T')\{T\} &= (\texttt{fun}\ H\{T\}\ T') \\
(\texttt{fun}\ U\ H)\{T\} &= (\texttt{fun}\ U\ H\{T\}) \\
(\texttt{con}\ \hat{\kappa}\ \vec{U}\ H\ \vec{T'})\{T\} &= (\texttt{con}\ \hat{\kappa}\ \vec{U}\ H\{T\}\ \vec{T'}) \\
(\texttt{comp}\ H)\{T\} &= (\texttt{comp}\ H\{T\}) \\
(\texttt{forall}\ \alpha\ K\ H)\{T\} &= (\texttt{forall}\ \alpha\ K\ H\{T\}) \\
[H\ T']\{T\} &= [H\{T\}\ T'] \\
[U\ H]\{T\} &= [U\ H\{T\}]
\end{aligned}
$$

Fig. 11. Type Context Insertion

$\boxed{T\ \rightarrow_{ty}^{*}\ U}$

Type $T$ reduces to type value U in some number of steps

$$\frac{}{U\ \rightarrow_{ty}^{*}\ U} \qquad \frac{T\ \rightarrow_{ty}\ T' \qquad T'\ \rightarrow_{ty}^{*}\ U}{T\ \rightarrow_{ty}^{*}\ U}$$

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

Type $T$ reduces in one step to type $T'$

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

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

Type $T$ locally reduces to type $T'$

$$\frac{}{[\,(\texttt{lam}\ \alpha\ K\ T)\ U\,]\ \Rightarrow_{ty}\ [U/\alpha]T}$$

Fig. 12. Type Reduction via Contextual Dynamics

| | | | | |
|---|---|---|---|---|
| Ctx | $E$ | ::= | $\circ$ | hole |
| | | | $(\texttt{isa}\ T\ E)$ | left annotation |
| | | | $(\texttt{inst}\ E\ T)$ | left instantiation |
| | | | $[E\ M]$ | left app |
| | | | $[V\ E]$ | right app |
| | | | $(\texttt{con}\ \hat{c}\ V^*\ E\ M^*)$ | condata |
| | | | $(\texttt{case}\ E\ C^*)$ | case |
| | | | $(\texttt{success}\ E)$ | success |
| | | | $(\texttt{bind}\ E\ x\ M)$ | bind |
| | | | $(\texttt{builtin}\ bi\ V^*\ E\ M^*)$ | builtin |

Fig. 13. Grammar of Reduction Contexts

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

Fig. 14. Context Insertion

$\boxed{M\ \rightarrow_\delta^*\ V}$

Term $M$ reduces in some number of steps to value $V$ in declaration environment $\delta$

$$\frac{}{V\ \rightarrow_\delta^*\ V}$$

$$\frac{M\ \rightarrow_\delta\ M' \qquad M'\ \rightarrow_\delta^*\ V}{M\ \rightarrow_\delta^*\ V}$$

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

Term $M$ reduces in one step to term $M'$ in declaration environment $\delta$

$$\frac{M\ \Rightarrow_\delta\ M'}{E\{M\}\ \rightarrow_\delta\ E\{M'\}}$$

Fig. 15. Reduction via Contextual Dynamics

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

Term $M$ locally reduces to term $M'$ in declaration context $\delta$

$$\frac{}{n\ \Rightarrow_{\delta, n\mapsto V}\ V}$$

$$\frac{}{(\texttt{isa}\ T\ M)\ \Rightarrow_\delta\ M}$$

$$\frac{}{(\texttt{inst}\ (\texttt{abs}\ \alpha\ M)\ T)\ \Rightarrow_\delta\ [T/\alpha]M}$$

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

$$\frac{\hat{c}, \vec{V}\ \sim\ \vec{C}\ \triangleright\ M}{(\texttt{case}\ (\texttt{con}\ \hat{c}\ \vec{V})\ \vec{C})\ \Rightarrow_\delta\ M}$$

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

Fig. 16. Local Reduction

$\boxed{\hat{c}, \vec{V} \ \sim \ \vec{C} \ \triangleright \ M}$

Constructor $\hat{c}$ with arguments $\vec{V}$ matches clauses $\vec{C}$ to produce result $M$

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

$$\frac{\hat{c} \ \neq \ \hat{c}' \qquad \hat{c}, \vec{V} \ \sim \ \vec{C} \ \triangleright \ M}{\hat{c}, \vec{V} \ \sim \ (\hat{c}' \ (\vec{x}) \ M'), \vec{C} \ \triangleright \ M}$$

Fig. 17. Case Matching

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

Fig. 18. Return Values

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

Instruction $I$ executes in some number of steps to return value $R$ in declaration environment $\delta$ and blockchain environment $E$

$$\frac{I \ \rightarrow^*_\delta \ (\texttt{success}\ V)}{I \ \rightsquigarrow^*_{E,\delta} \ (\texttt{ok}\ V)}$$

$$\frac{I \ \rightarrow^*_\delta \ (\texttt{failure})}{I \ \rightsquigarrow^*_{E,\delta} \ \texttt{err}}$$

$$\frac{I \ \rightarrow^*_\delta \ (\texttt{compbuiltin}\ cbi)}{I \ \rightsquigarrow^*_{E,\delta} \ (\texttt{ok}\ E_{cbi})}$$

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

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

Fig. 19. Execution

$$\begin{array}{rrcll} \text{Frame} & f & ::= & (\texttt{isa}\ T\ \_) & \text{left annotation} \\ & & & (\texttt{inst}\ \_\ T) & \text{left instantiation} \\ & & & [\_\ M] & \text{left app} \\ & & & [V\ \_] & \text{right app} \\ & & & (\texttt{con}\ \hat{c}\ V^*\ \_\ M^*) & \text{condata} \\ & & & (\texttt{case}\ \_\ C^*) & \text{case} \\ & & & (\texttt{success}\ \_) & \text{success} \\ & & & (\texttt{bind}\ \_\ x\ M) & \text{bind} \\ & & & (\texttt{builtin}\ bi\ V^*\ \_\ M^*) & \text{builtin} \\ \text{Stack} & s & ::= & f^* & \text{stacks} \\ \text{State} & \sigma & ::= & s \triangleright M & \text{computing} \\ & & & s \triangleleft V & \text{returning} \end{array}$$

Fig. 20. Grammar of CK Machine States

$\boxed{\sigma \;\mapsto_\delta\; \sigma'}$

...

$$
\begin{array}{l}
s \triangleright n \;\mapsto_{\delta, n \mapsto M}\; s \triangleright M \\
s \triangleright (\texttt{isa}\ T\ M) \;\mapsto_\delta\; s, (\texttt{isa}\ T\ \_) \triangleright M \\
s \triangleright (\texttt{abs}\ \alpha\ M) \;\mapsto_\delta\; s \triangleleft (\texttt{abs}\ \alpha\ M) \\
s \triangleright (\texttt{inst}\ M\ T) \;\mapsto_\delta\; s, (\texttt{inst}\ \_\ T) \triangleright M \\
s \triangleright (\texttt{lam}\ x\ M) \;\mapsto_\delta\; s \triangleleft (\texttt{lam}\ x\ M) \\
s \triangleright [M\ M'] \;\mapsto_\delta\; s, [\_\ M'] \triangleright M \\
s \triangleright (\texttt{con}\ \hat{c}\ \epsilon) \;\mapsto_\delta\; s \triangleleft (\texttt{con}\ \hat{c}\ \epsilon) \\
s \triangleright (\texttt{con}\ \hat{c}\ M\overline{M}) \;\mapsto_\delta\; s, (\texttt{con}\ \hat{c}\ \epsilon\ \_\ \overline{M}) \triangleright M \\
s \triangleright (\texttt{case}\ M\ \overline{C}) \;\mapsto_\delta\; s, (\texttt{case}\ \_\ \overline{C}) \triangleright M \\
s \triangleright (\texttt{success}\ M) \;\mapsto_\delta\; s, (\texttt{success}\ \_) \triangleright M \\
s \triangleright (\texttt{failure}) \;\mapsto_\delta\; s \triangleleft (\texttt{failure}) \\
s \triangleright (\texttt{compbuiltin}\ cbi) \;\mapsto_\delta\; s \triangleleft (\texttt{compbuiltin}\ cbi) \\
s \triangleright (\texttt{bind}\ M\ x\ M') \;\mapsto_\delta\; s, (\texttt{bind}\ \_\ x\ M') \triangleright M \\
s \triangleright (\texttt{builtin}\ bi\ \epsilon) \;\mapsto_\delta\; s \triangleleft (\texttt{builtin}\ bi\ \epsilon) \\
s \triangleright (\texttt{builtin}\ bi\ M\overline{M}) \;\mapsto_\delta\; s, (\texttt{builtin}\ bi\ \epsilon\ \_\ \overline{M}) \triangleright M \\
s, (\texttt{isa}\ T\ \_) \triangleleft V \;\mapsto_\delta\; s \triangleleft V \\
s, (\texttt{inst}\ \_\ T) \triangleleft (\texttt{abs}\ \alpha\ M) \;\mapsto_\delta\; s \triangleright [T/\alpha]M \\
s, [\_\ M'] \triangleleft V \;\mapsto_\delta\; s, [V\ \_] \triangleright M' \\
s, [\ (\texttt{lam}\ x\ M)\ \_] \triangleleft V \;\mapsto_\delta\; s \triangleright [V/x]M \\
s, (\texttt{con}\ \hat{c}\ \overline{V}\ \_\ \epsilon) \triangleleft V \;\mapsto_\delta\; s \triangleleft (\texttt{con}\ \hat{c}\ \overline{V}V) \\
s, (\texttt{con}\ \hat{c}\ \overline{V}\ \_\ M\overline{M}) \triangleleft V \;\mapsto_\delta\; s, (\texttt{con}\ \hat{c}\ \overline{V}V\ \_\ \overline{M}) \triangleright M \\
s, (\texttt{case}\ \_\ \overline{C}) \triangleleft (\texttt{con}\ \hat{c}\ \overline{V}) \;\mapsto_\delta\; s \triangleright \text{case on } (\texttt{con}\ \hat{c}\ \overline{V}) \text{ of } \overline{C} \\
s, (\texttt{success}\ \_) \triangleleft V \;\mapsto_\delta\; s \triangleleft (\texttt{success}\ V) \\
s, (\texttt{bind}\ \_\ x\ M) \triangleleft V \;\mapsto_\delta\; s \triangleleft (\texttt{bind}\ V\ x\ M) \\
s, (\texttt{builtin}\ bi\ \overline{V}\ \_\ \epsilon) \triangleleft V \;\mapsto_\delta\; s \triangleleft bi \text{ on } \overline{V}V \\
s, (\texttt{builtin}\ bi\ \overline{V}\ \_\ M\overline{M}) \triangleleft V \;\mapsto_\delta\; s, (\texttt{builtin}\ bi\ \overline{V}V\ \_\ \overline{M}) \triangleright M
\end{array}
$$

Fig. 21. CK Machine

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

Fig. 22. Builtin Types and Reductions

| *Comp Builtin Name* | *Result* |
|---|---|
| txhash | `(bytestring)` |
| blocknum | `(integer)` |
| blocktime | `(con Prelude.DateTime)` |

Fig. 23. Comp Builtin Types