Formal Specification of the Plutus Language

Darryl McAdams
Email: darryl.mcadams@iohk.io
Slack: @darryl.mcadams

I. Introduction

In this document, we outline the Plutus Language's overall design and give its big-step semantics. We also discuss high-level design considerations of the language, including the distinction between the Core language and the user-facing language. The latter language is accompanied by a proof theoretic definition of the elaboration and bidirectional typing process.

II. Plutus

The Plutus language is a variant of the λ calculus with functions and user-defined recursive data types, as well as a variety of build-in functions for cryptographic calculations. The language is split into two main parts, similar to Haskell: Plutus proper, which is the user-facing language that has convenient ways of declaring data and values, and a core language (Plutus Core, or just or Core for short), which is the actual language that defines the programs to run after Plutus is elaborated into it. The reason for this distinction will be discussed in detail later.

A. Plutus Core

Plutus Core is an untyped λ calculus. The grammar of the language is given in Figure 1, using the conventions of Harper [1]. 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 lists of terms, declared names, success and failure terms, and built-in functions. Terms live within a top-level program which consists of some declarations.

We do not bother to give an example program in Plutus Core here, as the abstract binding tree notation is not reader-friendly, nor is it intended to be read by humans.

The execution of a program in Plutus Core does not in itself result in any evaluation. Instead, the declarations are bound to their appropriate names. Then, designated names can be chosen to be evaluated in the declaration environment generated by the declarations. For instance, we might designate the name main to be the name who's definition we evaluated, as is done in Haskell. The generation of this declaration environment is given by the relation $D^* \gg \delta$, defined in Figure 2.

To give the computation rules for Plutus Core, we must define what the values are of the language, as given in Figure 3. These then let us define a parameterized binary relation $M \Downarrow_\delta M'$ which means M evaluates to M' using declarations δ, in Figure 4.

Tm	m	::=	x	variable
			$\texttt{decname}[\mathrm{n}]$	declared name
			$\texttt{let}(m;x.m)$	local declaration
			$\lambda(x.m)$	λ abstraction
			$\texttt{app}(m;m)$	function application
			$\texttt{con}[\mathrm{n}](m^*)$	constructed data
			$\texttt{case}(m^*;c^*)$	case
			$\texttt{success}(m)$	success
			$\texttt{failure}$	failure
			$\texttt{bind}(m;x.m)$	computation bind
			$\texttt{builtin}[\mathrm{n}](m^*)$	built-in function
Cl	c	::=	$\texttt{cl}(p^*;m)$	case clause
Pat	p	::=	x	variable pattern
			$\texttt{con}[\mathrm{n}](p^*)$	constructor pattern
Prg	g	::=	$\texttt{program}(d^*)$	program
Dec	d	::=	$\texttt{dec}(n;m)$	name declaration

Fig. 1. Grammar of Plutus Core

$\boxed{D \gg \delta}$

Declarations D generate environment δ

$$\frac{}{\epsilon \gg \epsilon}$$

$$\frac{D \gg \delta \qquad M \Downarrow_\delta M'}{D, \texttt{dec}(n;M) \gg \delta, n \mapsto M'}$$

Fig. 2. Declaration Environment Generation

Evaluation of case terms is defined in terms of pattern matching, given in Figure 5. Note that in the definition of evaluation for case terms, we assume that $\texttt{cl}(\vec{P};N)$ is the first clause of the case term for which $\vec{M'}$ matches the pattern.

Note also 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.

B. Plutus

Plutus proper, the user-facing language, is a strictly-typed, parametrically polymorphic language similar to Haskell and ML. It provides a convenient, user-friendly way to program, that can be elaborated down into Plutus Core. Plutus proper does not actually factor into any computation; it can be seen

$$
\begin{array}{llll}
\text{Val} & v & ::= & \lambda(x.m) \qquad \lambda \text{ abstraction} \\
& & & \texttt{con}[\text{n}](v^*) \qquad \text{constructed data} \\
& & & \texttt{success}(v) \qquad \text{success} \\
& & & \texttt{failure} \qquad \text{failure}
\end{array}
$$

Fig. 3. Values of Plutus Core

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

M evaluates to M' in environment δ

$$\frac{}{\texttt{decname}[\text{n}] \Downarrow_{\delta, \text{n} \mapsto M} M}$$

$$\frac{M \Downarrow_\delta M' \qquad [M'/x]N \Downarrow_\delta N'}{\texttt{let}(M;x.N) \Downarrow_\delta N'}$$

$$\frac{}{\lambda(x.M) \Downarrow_\delta \lambda(x.M)}$$

$$\frac{M \Downarrow_\delta \lambda(x.M') \qquad N \Downarrow_\delta N' \qquad [N'/x]M' \Downarrow_\delta M''}{\texttt{app}(M;N) \Downarrow_\delta M''}$$

$$\frac{M_i \Downarrow_\delta M_i'}{\texttt{con}[\text{n}](M_0, \ldots, M_n) \Downarrow_\delta \texttt{con}[\text{n}](M_0', \ldots, M_n')}$$

$$\frac{M_i \Downarrow_\delta M_i' \qquad M_i' \sim P_i \triangleright \sigma_i \qquad [\vec{\sigma}]N \Downarrow_\delta N'}{\texttt{case}(M_0, \ldots, M_k; \ldots, \texttt{cl}(P_0, \ldots, P_k; N), \ldots) \Downarrow_\delta N'}$$

$$\frac{M \Downarrow_\delta M'}{\texttt{success}(M) \Downarrow_\delta \texttt{success}(M')}$$

$$\frac{}{\texttt{failure} \Downarrow_\delta \texttt{failure}}$$

$$\frac{M \Downarrow_\delta \texttt{success}(M') \qquad [M'/x]N \Downarrow_\delta N'}{\texttt{bind}(M;x.N) \Downarrow_\delta N'}$$

$$\frac{M \Downarrow_\delta \texttt{failure}}{\texttt{bind}(M;x.N) \Downarrow_\delta \texttt{failure}}$$

$$\frac{M_i \Downarrow_\delta M_i' \qquad \textit{built-in } \text{n} \textit{ evals on } M_0', \ldots, M_n' \textit{ to } N}{\texttt{builtin}[\text{n}](M_0, \ldots, M_n) \Downarrow_\delta N}$$

Fig. 4. Evaluation of Plutus Core

$\boxed{M \sim P \triangleright \sigma}$

M matches pattern P yielding substitutions σ

$$\frac{}{M \sim x \triangleright M/x}$$

$$\frac{M_i \sim P_i \triangleright \sigma_i}{\texttt{con}[\text{n}](M_0, \ldots, M_n) \sim \texttt{con}[\text{n}](P_0, \ldots, P_n) \triangleright \sigma_0, \ldots, \sigma_n}$$

Fig. 5. Pattern Matching in Plutus Core

Tm	m	::=	x	variable
			n	declared name
			$m : t$	type annotation
			$\texttt{let}\ \{\ tmd^*\ \}\ \texttt{in}\ m$	local declarations
			$\lambda x \to m$	λ abstraction
			$m\ m$	function application
			$\text{n}\ m^*$	constructed data
			$\texttt{case}\ m^*\ \texttt{of}\ \{\ c^*\ \}$	case
			$\texttt{success}\ m$	success
			$\texttt{failure}$	failure
			$\texttt{do}\ \{\ x \leftarrow m\ ;\ m\ \}$	do notation
			$!\text{n}\ m^*$	built-in function
Cl	c	::=	$p^* \to m$	case clause
Pat	p	::=	x	variable pattern
			$\text{n}\ p^*$	constructor pattern
Prg	g	::=	d^*	programs
Dec	d	::=	tyd	type declaration
			tmd	term declaration
TyDec	tyd	::=	$\texttt{data}\ \text{n}\ \alpha^* = tyc^*$	type declaration
TyCl	tyc	::=	$\text{n}\ t^*$	type alternative clause
Ty	t	::=	α	type variable
			$\text{n}\ t^*$	type constructor
			$t \to t$	function type
			$\forall \alpha.\ t$	polymorphic quantifier
			$\text{Comp}\ t$	computations
TmDec	tmd	::=	$\text{n} : t\ \{\ tmc^*\ \}$	term declaration
TmCl	tmc	::=	$\text{n}\ p^* = m$	term clause

Fig. 6. Grammar of Plutus

entirely as a tool for writing Plutus Core programs. We will discuss this topic more in the next section.

Plutus proper is defined by the grammar given in Figure 6. This includes definitions for the types of Plutus, as well as terms and declarations (of both types and terms). The grammar is intentionally Haskell-like. Blocks that should be parsed via indentation are specified with braces. Additionally, we assume that *let*, λ and *do* terms, and $\forall$ types, are sugarable as in Haskell to have iterated bindings. Finally, while the grammar permits pattern matching definitions of functions, we treat term declarations in the elaborator as if they only permit defining a name to be a single term. We do this because the former can be transformed into the latter in conventional ways, thus simplifying things greatly.

An example program in Plutus is the canonical factorial function over Peano numerals. First we must define the relevant types, and basic functions, and then we can define the factorial function. This is shown in Figure 7.

The statics for Plutus can be given in the form of a collection of elaboration judgments. Whole program elaboration uses the judgment $\Sigma; \Delta \vdash P \dashv \Sigma'; \Delta'$, which says that a program P elaborates by transforming the signature Σ and declarations Δ into a new signature Σ' and new declarations Δ'.

A signature consists of a number of declarations, for both type constructor arity (of the form $\text{n} : \star^k$), where k is the number of type arguments that the type constructor n takes, and term constructor arity (of the form $\text{n} : [\vec{\alpha}](\vec{A})B$, where $\vec{\alpha}$ is a collection of type variables that the constructor's arguments and return type can refer to, $\vec{A}$ are the types of the arguments to the constructor n, and B is the type of the value that n constructors. So for example, the type constructor for natural numbers would be in a signature as $\text{Nat} : \star^0$, with 0 type arguments because it's a nullary type constructor, while list

$$\texttt{data}\ \text{Nat} = \text{Zero} \mid \text{Suc Nat}$$

$$\begin{aligned}
&\text{plus} : \text{Nat} \to \text{Nat} \to \text{Nat}\\
&\text{plus Zero}\ \ n = n\\
&\text{plus (Suc } m)\ n = \text{Suc (plus } m\ n)
\end{aligned}$$

$$\begin{aligned}
&\text{times} : \text{Nat} \to \text{Nat} \to \text{Nat}\\
&\text{times Zero}\ \ n = \mathit{Zero}\\
&\text{times (Suc } m)\ n = \text{plus } n\ \text{(times } m\ n)
\end{aligned}$$

$$\begin{aligned}
&\text{fac} : \text{Nat} \to \text{Nat}\\
&\text{fac Zero}\ \ = \text{Suc Zero}\\
&\text{fac (Suc } n) = \text{times (Suc } n)\ \text{(fac } n)
\end{aligned}$$

Fig. 7. Fibonacci in Plutus

$$\boxed{\Sigma; \Delta \ \vdash\ P\ \dashv\ \Sigma'; \Delta'}$$

Program P elaborates by transforming signature Σ and declarations Δ into new signature Σ' and new declarations Δ'

$$\frac{}{\Sigma; \Delta \ \vdash\ \epsilon\ \dashv\ \Sigma; \Delta}$$

$$\frac{\begin{array}{c}\Sigma \ \vdash\ \mathit{type}\ \text{n}\ \vec{\alpha}\ \mathit{alts}\ \vec{C}\ \dashv\ \Sigma'\\ \Sigma'; \Delta \ \vdash\ P\ \dashv\ \Sigma''; \Delta''\end{array}}{\Sigma; \Delta \ \vdash\ \texttt{data}\ \text{n}\ \vec{\alpha} = \vec{C};\ P\ \dashv\ \Sigma''; \Delta''}$$

$$\frac{\begin{array}{c}\Sigma; \Delta \ \vdash\ \mathit{term}\ \text{n}\ \mathit{type}\ A\ \mathit{def}\ M\ \dashv\ \Delta'\\ \Sigma; \Delta' \ \vdash\ P\ \dashv\ \Sigma''; \Delta''\end{array}}{\Sigma; \Delta \ \vdash\ \text{n} : A\ \{\ M\ \};\ P\ \dashv\ \Sigma''; \Delta''}$$

Fig. 8. Elaboration of Programs in Plutus

would be List : $\star^1$, with one type argument (the parameter for its element values). Their respective constructors would be in a signature as zero : $[]()\text{Nat}$, suc : $[](\text{Nat})\text{Nat}$, nil : $[\alpha]()(\text{List } \alpha)$, and cons : $[\alpha](\alpha, \text{List } \alpha)(\text{List } \alpha)$.

Declaration Δ consist of entries of the form $\text{n} : A \mapsto M'$, where n is the name of a declared term, A is its type, and M' is its definition, which is a term in Plutus Core.

The overall elaboration process uses the judgments to turn a set of type and term declarations into a signature and set of declared values that define a program. These judgments are defined in Figure 8. The essence of program elaboration is that we just elaborate each declaration, in sequence, threading the signature and declarations through each sub-elaboration to accumulate the results.

Naturally, we now need to explain how type declarations and term declarations are elaborated. Type declarations will simply elaborate each alternative for the type, adding them to the signature, along with the arity of the whole type constructor. This is shown in Figure 9.

Elaboration of term declarations is simpler, requiring only that we check that the type is valid and that the definition has that type, as shown in Figure 10. This results in the extension of the declarations with the new term.

$$\boxed{\Sigma \ \vdash\ \mathit{type}\ \text{n}\ \vec{\alpha}\ \mathit{alts}\ \vec{C}\ \dashv\ \Sigma'}$$

Type declaration for n with parameters $\vec{\alpha}$ and alternatives $\vec{C}$ elaborates by transforming signature Σ into new signature Σ'

$$\frac{\begin{array}{c}\Sigma, \text{n} : \star^k \ \vdash\ C_0\ \mathit{alt}\ \text{n}\ \vec{\alpha}\ \dashv\ \Sigma'_0\\ \Sigma'_0 \ \vdash\ C_1\ \mathit{alt}\ \text{n}\ \vec{\alpha}\ \dashv\ \Sigma'_1\\ \cdots\\ \Sigma'_{j-1} \ \vdash\ C_j\ \mathit{alt}\ \text{n}\ \vec{\alpha}\ \dashv\ \Sigma'_j\end{array}}{\Sigma \ \vdash\ \mathit{type}\ \text{n}\ \vec{\alpha}\ \mathit{alts}\ C_0 \mid \ \ldots\ \mid C_j\ \dashv\ \Sigma'_j}$$

$$\boxed{\Sigma \ \vdash\ \text{c}\ \vec{A}\ \mathit{alt}\ \text{n}\ \vec{\alpha}\ \dashv\ \Sigma'}$$

Constructor c with arg types $\vec{A}$ elaborates for type constructor n with parameters $\vec{\alpha}$ by transforming signature Σ into new signature Σ'

$$\frac{\Sigma; \vec{\alpha}\ \mathit{type} \ \vdash\ A_i\ \mathit{type}}{\Sigma \ \vdash\ \text{c}\ A_0\ \ldots\ A_k\ \mathit{alt}\ \text{n}\ \vec{\alpha}\ \dashv\ \Sigma, \text{c} : [\vec{\alpha}](A_0, \ldots, A_k)(\text{n}\ \vec{\alpha})}$$

Fig. 9. Elaboration of Type Declarations in Plutus

$$\boxed{\Sigma; \Delta \ \vdash\ \mathit{term}\ \text{n}\ \mathit{type}\ A\ \mathit{def}\ M\ \dashv\ \Delta'}$$

Term declaration for decname[n] with type A and definition M elaborates by using signature Σ to transform declarations Δ into new declarations Δ'

$$\frac{\Sigma \ \vdash\ A\ \mathit{type} \qquad \Sigma\ ;\ \Delta\ ;\ \text{n} : A \ \vdash\ A\ \ni\ M\ \triangleright\ M'}{\Sigma; \Delta \ \vdash\ \mathit{term}\ \text{n}\ \mathit{type}\ A\ \mathit{def}\ M\ \dashv\ \Delta, \text{n} : A \mapsto M'}$$

Fig. 10. Elaboration of Term Declarations in Plutus

As can be seen in term declaration elaboration, there are also term-level elaboration judgments. In particular, we have two typing judgments, for checking and for synthesizing types: $\Sigma\ ;\ \Delta\ ;\ \Gamma\ \vdash\ T\ \ni\ M\ \triangleright\ M'$, which says that with signature Σ, declarations Δ, and in the context Γ, the type T contains the Plutus term M which elaborates to Core term M', and $\Sigma\ ;\ \Delta\ ;\ \Gamma\ \vdash\ M\ \triangleright\ M'\ \in\ T$, which says that with signature Σ, declarations Δ and in context Γ, the term M elaborates to M', synthesizing the type T. These are given in Figure 11 and Figure 12. Since this gets very verbose, Σ, Δ, and Γ will be suppressed except when they different between premises and conclusion, or when they are relevant to the particular rule at hand.

A number of auxiliary judgments are used in the definition of elaboration. The judgment $A\ \mathit{type}$, which is short for $\Sigma; \Gamma\ \vdash\ A\ \mathit{type}$, is the judgment that A is a type in signature Σ and context Γ. The judgment $\vec{P} \to M\ \triangleright\ M'\ \mathit{from}\ \vec{A}\ \mathit{to}\ B$, which is short for $\Sigma; \Delta; \Gamma\ \vdash\ \vec{P} \to M\ \triangleright\ M'\ \mathit{from}\ \vec{A}\ \mathit{to}\ B$, is the judgment that $\vec{P} \to M$ is a clause with patterns for types $\vec{A}$ returning a body with type B which elaborates to M', in signature Σ, declarations Δ, and context Γ. This is in turn defined in terms of a judgment $\Sigma\ \vdash\ P\ \mathit{pattern}\ A\ \dashv\ \Gamma'$, which is the judgment that P is a pattern for A in signature Σ that binds variables Γ'. Finally, the judgment $A\ \sqsubseteq\ B$ defines a subtyping judgment used for instantiating quantifiers. These auxiliary judgments are defined in Figure 13. We do not bother

$$\boxed{\Sigma\ ;\ \Delta\ ;\ \Gamma\ \vdash\ A\ \ni\ M\ \triangleright\ M'}$$

Type A contains term M which elaborates to M' in signature Σ, declarations Δ, and context Γ

$$\frac{\begin{array}{c}\Gamma\ \vdash\ B\ \mathit{type} \\ \Gamma\ \vdash\ B\ \ni\ M\ \triangleright\ M' \\ \Gamma, x : B\ \vdash\ A\ \ni\ N\ \triangleright\ N'\end{array}}{\Gamma\ \vdash\ A\ \ni\ \texttt{let}\ \{\ x : B\ \{\ M\ \}\ \}\ \texttt{in}\ N\ \triangleright\ \texttt{let}(M';x.N')}$$

$$\frac{\Gamma, x : A\ \vdash\ B\ \ni\ M\ \triangleright\ M'}{\Gamma\ \vdash\ A \to B\ \ni\ \lambda x \to M\ \triangleright\ \lambda(x.M')}$$

$$\frac{\begin{array}{c}\Sigma\ \ni\ \mathrm{n}\ :\ [\vec{\alpha}](A_0, \ldots, A_n)B \\ [\sigma]B = B' \\ \Sigma\ \vdash\ [\sigma]A_i\ \ni\ M_i\ \triangleright\ M_i'\end{array}}{\Sigma\ \vdash\ B'\ \ni\ \mathrm{n}\ M_0\ \ldots\ M_n\ \triangleright\ \texttt{con}[\mathrm{n}](M_0'; \ldots; M_n')}$$

$$\frac{A\ \ni\ M\ \triangleright\ M'}{\mathrm{Comp}\ A\ \ni\ \texttt{success}\ M\ \triangleright\ \texttt{success}(M')}$$

$$\frac{}{\mathrm{Comp}\ A\ \ni\ \texttt{failure}\ \triangleright\ \texttt{failure}}$$

$$\frac{\begin{array}{c}\Gamma\ \vdash\ M\ \triangleright\ M'\ \in\ \mathrm{Comp}\ A \\ \Gamma, x : A\ \vdash\ \mathrm{Comp}\ B\ \ni\ N\ \triangleright\ N'\end{array}}{\Gamma\ \vdash\ \mathrm{Comp}\ B\ \ni\ \texttt{do}\ \{\ x \leftarrow M\ ;\ N\ \}\ \triangleright\ \texttt{bind}(M';x.N')}$$

$$\frac{\Gamma, \alpha\ \mathit{type}\ \vdash\ A\ \ni\ M\ \triangleright\ M'}{\Gamma\ \vdash\ \forall\alpha.\ A\ \ni\ M\ \triangleright\ M'}$$

$$\frac{M\ \triangleright\ M'\ \in\ A \qquad A\ \sqsubseteq\ B}{B\ \ni\ M\ \triangleright\ M'}$$

Fig. 11. Checking Judgment

to define the judgments $\Sigma\ \ni\ \mathrm{n}\ :\ A$, $\Delta\ \ni\ \mathrm{n}\ :\ A$, and $\Gamma\ \ni\ x\ :\ A$ as they are relatively obvious and uninteresting.

III. Core vs. Non-Core

The choice to have a core language, Plutus Core, and a non-core language, Plutus, provides a number of benefits. Plutus Core is simpler, and so formal verification of behavior is easier. Additionally, it's simplicity means that it can be more easily serialized. But the main benefit of having Core be separate from Plutus proper is that Plutus proper is not actually part of the blockchain protocol. In effect, Plutus is just a tool for editing Plutus Core. If it becomes desirable to have a language with a more powerful collection of tools, such as type classes and GADTs, or even dependent types, they can be layered on top of Core as another distinct way to use Core, without having to make any changes to the protocol's scripting language at all. The syntax and semantics of Core remains unchanged, while these additions are freely swappable, because they all elaborate down to Core.

This also makes it possible to do interesting things with embedded domain specific languages. Because Core has functions and data constructors, it is possible to define DSLs that layer on top of Core as a special mode of use. Consider, for instance,

$$\boxed{\Sigma\ ;\ \Delta\ ;\ \Gamma\ \vdash\ M\ \triangleright\ M'\ \in\ A}$$

Term M elaborates to M' synthesizing type A in signature Σ, declarations Δ, and context Γ

$$\frac{\Gamma\ \ni\ x\ :\ A}{\Gamma\ \vdash\ x\ \triangleright\ x\ \in\ A}$$

$$\frac{\Delta\ \ni\ \mathrm{n}\ :\ A}{\Delta\ \vdash\ \mathrm{n}\ \triangleright\ \texttt{decname}[\mathrm{n}]\ \in\ A}$$

$$\frac{A\ \mathit{type} \qquad A\ \ni\ M\ \triangleright\ M'}{M : A\ \triangleright\ M'\ \in\ A}$$

$$\frac{M\ \triangleright\ M'\ \in\ A \to B \qquad A\ \ni\ N\ \triangleright\ N'}{M\ N\ \triangleright\ \texttt{app}(M';N')\ \in\ B}$$

$$\frac{\begin{array}{c}M_i\ \triangleright\ M_i'\ \in\ A_i \qquad \vec{P_j} \to N_j\ \triangleright\ N_j'\ \mathit{from}\ A_0, \ldots, A_m\ \mathit{to}\ B \\ \hline \texttt{case}\ M_0\ \mid\ \ldots\ \mid\ M_m\ \texttt{of}\ \{\ \vec{P_0} \to N_0; \ldots; \vec{P_n} \to N_n\ \}\ \triangleright \\ \texttt{case}(M_0', \ldots, M_m'; \texttt{cl}(\vec{P_0}; N_0'), \ldots, \texttt{cl}(\vec{P_n}; N_n'))\ \in\ B\end{array}}{}$$

$$\frac{\begin{array}{c}\Sigma\ \ni\ \mathrm{n}\ :\ [\vec{\alpha}](A_0, \ldots, A_k)B \\ [\sigma]B = B' \\ \Sigma\ \vdash\ [\sigma]A_i\ \ni\ M_i\ \triangleright\ M_i'\end{array}}{\Sigma\ \vdash\ !\mathrm{n}\ M_0\ \ldots\ M_k\ \triangleright\ \texttt{builtin}[\mathrm{n}](M_0'; \ldots; M_k')\ \in\ B'}$$

Fig. 12. Synthesis Judgment

the MAST proposal from Bitcoin. While Core itself is likely impossible to use for MAST because of higher order functions, general recursion, and declared values, it is entirely possible to define a special data type and evaluation function that encodes a MAST computation. In this way, MAST becomes just a special little library in Core. Using the restricted functionality of MAST would be entirely optional, and the full power of Core would still be available for aspects of the program which do not need to be hidden or compressed.

Another benefit to using Core, instead of a lower-level language, is that it leaves the implementation of the language somewhat free. Any implementation can perfectly well choose to use Core directly, by using an evaluator in the way the standard semantics does, or it can use a virtual machine of some sort. Two such machines are given below, one a CEK machine and the other a stack machine.

IV. Basic Validation Program Structure

The basic way that validation is done in Plutus 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, we have special constructs for this.

In particular, the `success` e and `failure` constructs in the $Comp$ type. Any program which validates a transaction must declare a function $\mathrm{validator} : A \to \mathrm{Comp}\ B$ for some A and B, while the corresponding program supplied by the redeemer must declare $\mathrm{redeemer} : \mathrm{Comp}\ A$. 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 evaluating the term

$\boxed{\Sigma; \Gamma \ \vdash \ A \ type}$

A is a type using signature Σ and context Γ

$$\frac{}{\Gamma, \alpha \ type \ \vdash \ \alpha \ type}$$

$$\frac{A \ type \qquad B \ type}{A \to B \ type}$$

$$\frac{\Sigma \ \ni \ \mathrm{n} \ : \ \star^k \qquad \Sigma \ \vdash \ A_i \ type}{\Sigma \ \vdash \ \mathrm{n} \ A_0 \ \ldots A_k \ type}$$

$$\frac{\Gamma, \alpha \ type \ \vdash \ A \ type}{\Gamma \ \vdash \ \forall \alpha. \ A \ type}$$

$\boxed{A \ \sqsubseteq \ B}$

A is a subtype of B

$$\frac{A \ \sqsubseteq \ B}{A \ \sqsubseteq \ \forall \alpha. \ B}$$

$$\frac{[T/\alpha]A \ \sqsubseteq \ B}{\forall \alpha. \ A \ \sqsubseteq \ B}$$

$$\frac{A' \ \sqsubseteq \ A \qquad B \ \sqsubseteq \ B'}{A \to B \ \sqsubseteq \ A' \to B'}$$

$$\frac{}{A \ \sqsubseteq \ A}$$

$\boxed{\Sigma; \Delta; \Gamma \ \vdash \ \vec{P} \to M \ \triangleright \ M' \ \textit{from} \ \vec{A} \ \textit{to} \ B}$

$\vec{P} \to M$ is a clause whose body elaborates to M' with pattern types $\vec{A}$ and body type B, using signature Σ, declarations Δ, and context Γ

$$\frac{\begin{array}{c}\Sigma \ \vdash \ A_i \ \textit{pattern} \ P_i \ \dashv \ \Gamma_i' \\ \Sigma \ ; \ \Delta \ ; \ \Gamma, \Gamma_0', \ldots, \Gamma_k' \ \vdash \ B \ \ni \ M \ \triangleright \ M'\end{array}}{\Sigma; \Delta; \Gamma \ \vdash \ P_0 \mid \ \ldots \ \mid P_k \to M \ \triangleright \ M' \ \textit{from} \ A_0, \ldots, A_k \ \textit{to} \ B}$$

$\boxed{\Sigma \ \vdash \ A \ \textit{pattern} \ P \ \dashv \ \Gamma'}$

Type A has pattern P using signature Σ yielding context Γ'

$$\frac{}{\Sigma \ \vdash \ A \ \textit{pattern} \ x \ \dashv \ x : A}$$

$$\frac{\begin{array}{c}\Sigma \ \ni \ \mathrm{n} \ : \ [\vec{\alpha}](A_0, \ldots, A_n)B \\ [\sigma]B = B' \\ \Sigma \ \vdash \ A_i \ \textit{pattern} \ P_i \ \dashv \ \Gamma_i'\end{array}}{\Sigma \ \vdash \ B' \ \textit{pattern} \ \mathrm{n} \ P_0 \ \ldots \ P_k \ \dashv \ \Gamma_0', \ldots, \Gamma_k'}$$

Fig. 13. Auxuliary Elaboration Judgments of Plutus

`do` $\{ \ x \leftarrow \mathrm{redeemer} \ ; \ \mathrm{validator} \ x \ \}$ (or rather, the Plutus Core term that it corresponds to). If this evaluates to `success` e for some e, 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 evaluates to `failure`, then the transaction is invalid, analogous to Bitcoin Script either failing to execute, or leaving $false$ on the top of stack. The value returned in the success case is irrelevant by validation.

This approach to validation requires explicit control over the success or failure of the transaction. This is distinct from Bitcoin Script, where failure can occur because of some implicitly effectful operations that can immediately kill the entire execution. In Plutus, there are no effects, so any validation failure must be explicitly stated to be a failure using `failure`, as must any validation success with `success` e.

REFERENCES

[1] Harper, R. *Practical Foundations for Programming Languages*.