Simplicity

Russell O'Connor

Blockstream

Email: roconnor@blockstream.com

February 13, 2024

DRAFT

Table of contents

Chapter 1
Introduction

1.1 Bitcoin Script

Bitcoin[?] was the first protocol that used a blockchain to build a distributed ledger that allows anyone to transact a cryptographic currency with minimal risk of their transaction being reversed or undone, and without relying on a trusted third party or central authority. Typically access to funds are controlled by a cryptographic private key. References to one or more of these funds, which may or may not have the same private keys, are assembled into a data structure, called a *transaction*, along with a set of one or more outputs which specify which cryptographic public keys that will control each output. This transaction data is signed with each private key for each input and added to the transaction as *witness data*.

More precisely, the outputs are not necessarily controlled by a simple single cryptographic key, rather each output is controlled by a small program written in a language called *Bitcoin Script*[?, ?]. Bitcoin Script is a stack-based language with conditionals operations for control flow and no loops. Bitcoin has stack manipulation operations, Boolean logic operations, and very simple arithmetic operations (without even multiplication). It also has some cryptographic operations that include cryptographic hash functions, and digital signature verification operations. The `CHECKSIG` operation does an ECDSA digital signature verification of transaction data.

A basic example of a Bitcoin Script program pushes an ECDSA public key onto the stack, followed by a `CHECKSIG` operation. This Script is part of the output data within a transaction. In order to spend the funds held by such an output, one builds a transaction with an input referencing this output and adds a ECDSA signature to the transaction's witness data for the input (called a ScriptSig). The witness data describes which part of the transaction data is being signed (typically everything other than the witness data itself) and has the signature.

To validate an input, the witness data is pushed onto the stack, and the the program in the referenced output is executed. In our example above, the program pushes a specific ECDSA public key onto the stack and then executes `CHECKSIG`. The `CHECKSIG` operation pops the public key and the signature data off the stack. Then it cryptographically hashes the transaction data as specified by the signature data, including which input is being signed, and verifies that the digital signature for that public key is valid for that hashed data. Only if successful is the value `1` pushed back onto the stack. In order for a transaction to be valid, all inputs are checked in this fashion, by pushing its associated witness data onto the stack and then executing the program found in the referenced output and requiring a non-zero value be left on the stack.

From this example, we see that there are three parts that go into checking if a transaction's input is authorized to be spent:

- A Bitcoin Script, which is contained in a transaction output.
- Witness data, for each transactions input.
- The rest of the transaction data, which includes things like output amounts, version numbers, etc.

All the data needed to validate a transaction is part of (or cryptographically committed within) the transaction data. This means the inputs can be validated independently of the state of the rest of the blockchain, as far as Bitcoin Script is concerned.

In the above example, the Bitcoin Script is included into a transaction's output, at what I call *commitment time*. The signature and transaction data is specified later at what I call *redemption time*. That said, the Bitcoin Script does not have to be entirely presented at commitment time; it is acceptable to just commit to the Bitcoin Script's cryptographic hash. Then, at redemption time, the script is revealed together with the witness data and the rest of the transaction. In this case the program's hash is verified to be equal to the hash committed in the output, and then the witness data, and revealed Bitcoin Script are validated as before. Bitcoin's *pay to script hash* (a.k.a. *P2SH*) follows this modified procedure which makes it easy for users to specify complex Bitcoin Scripts with to control funds while only needing to provide a single hash value to their counterparty when they are first receiving their funds.

More complex scripts can be devised to do multi-signature, escrow, hashed-timelock contracts, etc. However, because of the the limited expressiveness of Bitcoin Script (e.g. no multiplication operation), only so much can be programmed. Therefore, we want to design an alternative to Bitcoin Script that will be more expressive, without sacrificing the good properties of Bitcoin Script. We call this new language *Simplicity*.

1.2 Simplicity's Design Goals

Our goals for Simplicity include the following:

- Create an expressive language that lets users build novel programs and smart contracts.
- Enable useful static analysis to bound computational resource usage of programs.
- Minimize bandwidth and storage requirements by allowing sharing of expressions and pruning of unused expressions.
- Capture the computational environment entirely within a unit of transaction.
- Provide formal semantics for reasoning with off-the-shelf proof-assistant software.

1.2.1 Static Analysis

In Bitcoin Script, static analysis lets us count how many operations there are and, in particular, bound the maximum number of expensive operations, such as signature validation, that could be performed. We can do this because Bitcoin Script has no looping operations. In a transaction, if Bitcoin Script is deemed to be potentially too expensive from this analysis, the program is rejected at redemption time.

We want to support this same sort of static analysis in Simplicity. It lets users determine, at commitment time, the worst case cost for redeeming the program, for any possible witness inputs. At redemption time it allows, as part of the consensus rules, to bound the cost of running programs prior to execution. This serves as one method of avoiding denial of service attacks in a blockchain protocol like Bitcoin.

1.2.2 Pruning and Sharing

As we saw before at commitment time we only need to specify a cryptographic commitment to the program, and the program can be revealed at redemption time. For more complex programs with multiple alternative branches, only those branches that are actually going to be executed need to be revealed, and the remainder of the program that is not executed for this particular transaction can be excluded.

Using a technique called *Merkelized Abstract Syntax Trees* a.k.a. *MAST*, we can commit to a Merkle root of a program's abstract syntax tree. At redemption time, we can prune unused sub-expressions. Instead only the Merkle root of the pruned branches need to be revealed in order to verify that the program's Merkle root matches the root specified at commitment time.

Because two identical subexpressions necessarily have the same Merkle root, this procedure also lets us reduce program size by sharing these identical subexpressions. In principle, this sharing could extend as wide as between different program in different transactions.

1.2.3 Formal Semantics

Once a program has been committed to, it cannot be changed or updated later. If a program has errors or security vulnerabilities, it can no longer be fixed. Therefore, it is essential to get these program correct. Fortunately these programs tend to be relatively small, and hence amenable to formal analysis. We use the Coq proof assistant [?] to specify formal semantics of Simplicity. This allows us to both reason about programs written in Simplicity, and also lets us reason about our static analysis and Simplicity interpreters and prove that they are correct.

While we specifically use the Coq proof assistant, the formal semantics of Simplicity is designed to be easy enough to define in any proof assistant, especially others based on dependent type theory.

Chapter 2
Type Theory Preliminaries

In this document we endeavour to be precise about the semantics surrounding the definition of the Simplicity language. To this end we use formal language composed from a mix of mathematics, category theory, simple type theory, and functional programming. While most all our notation is standard in some community, readers may not be familiar with all of it. To that end, and to ensure completeness of our semantics, we give detailed definitions below of the common notation used throughout this document.

Our formal language is phrased in terms of simple type theory and readers familiar with mathematics should not have too much trouble following it since mathematics notation already borrows heavily from simple type theory.

To begin with we will assume that we have a notation of a type of natural numbers, $\mathbb{N}$, with $0 : \mathbb{N}$, $1 : \mathbb{N}$, and so forth for all other numbers. Given natural numbers $n : \mathbb{N}$ and $m : \mathbb{N}$, we take it for granted that we have

- notions of arithmetic including $n + m$, $n\,m$, and n^m;
- comparison operations including $n < m$, $n \leq m$, $n \geq m$, and $n > m$;
- when $n \geq m$ then the difference $n - m$ is a natural number.

We will partake in several notational conveniences. We will generally elide parentheses when the value they surround already has has its own brackets. For example,

- we will write $f\langle x, y\rangle$ instead of $f(\langle x, y\rangle)$;
- we will write $f\texttt{[cafe]}$ instead of $f(\texttt{[cafe]})$;
- we will write $l\lceil n \rceil$ instead of $l[\lceil n \rceil]$.

As you will see, we have a lot of notation with type annotations that are used to fully disambiguate them. Often these type annotations can be completely inferred from the surrounding context and accordingly we will usually omit these annotations to reduce notational clutter and reserve the annotated versions for cases where the annotations are ambiguous or where we want to draw specific attention to them.

2.1 Algebraic Types

We write the primitive unit type as $\mathbb{1}$. The unique value of the unit type is $\langle\rangle : \mathbb{1}$.

Given types A and B, then $A + B$, $A \times B$, and $A \to B$ are the sum type (also known as disjoint union type), the (Cartesian) product type, and the function type respectively. Given $a : A$ and $b : B$ we denote values of the sum and product types as

$$\begin{aligned} \sigma^{\mathbf{L}}_{A,B}(a) &: A + B \\ \sigma^{\mathbf{R}}_{A,B}(b) &: A + B \\ \langle a, b\rangle_{A,B} &: A \times B \end{aligned}$$

We will usually omit the annotations, writing $\sigma^{\mathbf{L}}(a)$, $\sigma^{\mathbf{R}}(b)$, and $\langle a, b\rangle$.

We write an expression of type $A \to B$ using lambda notation:

$$\lambda x{:}\, A.\, e : A \to B$$

where e is an expression with $x : A$ as a bound variable (that may occur or zero or more times in e). Given $f : A \to B$ and $a : A$, then ordinary function application retrieves a value of type B:

$$f(a) : B$$

The type operator $\to$ is right associative:

$$A \to B \to C = A \to (B \to C)$$

We define the identity function $\mathrm{id}_A : A \to A$ as

$$\mathrm{id}_A := \lambda a\colon A.\, a$$

and given $f : A \to B$ and $g : B \to C$ we define their composition $g \circ f : A \to C$ as

$$g \circ f := \lambda a\colon A.\, g(f(a)).$$

We may also write function definitions in "application" style where we implicitly define a function in terms of function application. In this style we would write the above definitions of the identity function and function composition as

$$\begin{aligned} \mathrm{id}_A(a) &:= a \\ (g \circ f)(a) &:= g(f(a)). \end{aligned}$$

To access components of sum and product types we define functions using pattern matching in application style. For example given $a : A$ and $b : B$, we define the first and second projection functions as

$$\begin{aligned} \pi_1^{A,B}\langle a, b\rangle &:= a \\ \pi_2^{A,B}\langle a, b\rangle &:= b \end{aligned}$$

and given $f : A \to C$ and $g : B \to C$, we define their copair $[f, g] : A + B \to C$ by the pair of equations

$$\begin{aligned} [f, g](\sigma^{\mathbf{L}}(a)) &:= f(a) \\ [f, g](\sigma^{\mathbf{R}}(b)) &:= g(b). \end{aligned}$$

When we take a product type with itself, we form a square and denote it by exponential notation accordingly:

$$A^2 := A \times A$$

When we take repeated squares, we denote this by exponential notation with successively larger powers of two:

$$\begin{aligned} A^1 &:= A \\ A^2 &:= A^1 \times A^1 \\ A^4 &:= A^2 \times A^2 \\ &\vdots \\ A^{2^{1+n}} &:= A^{2^n} \times A^{2^n} \\ &\vdots \end{aligned}$$

We define the diagonal function returning a square type, $\Delta_A : A \to A^2$:

$$\Delta_A(a) := \langle a, a\rangle$$

We define $\mathbb{2}$ as the Boolean type (or Bit type):

$$\mathbb{2} := \mathbb{1} + \mathbb{1}$$

We name the two values of this type, $\mathtt{0}_{\mathbb{2}} : \mathbb{2}$ and $\mathtt{1}_{\mathbb{2}} : \mathbb{2}$.

$$\begin{aligned} \mathtt{0}_{\mathbb{2}} &:= \sigma^{\mathbf{L}}_{\mathbb{1},\mathbb{1}}\langle\rangle \\ \mathtt{1}_{\mathbb{2}} &:= \sigma^{\mathbf{R}}_{\mathbb{1},\mathbb{1}}\langle\rangle \end{aligned}$$

2.1.1 Records

Record types are essentially the same as product types, but with fancy syntax. We write a record type enclosed by curly braces.

$$\left\{\begin{array}{l}\text{field}_1 : A_1\\ \text{field}_2 : A_2\\ \vdots\\ \text{field}_n : A_n\end{array}\right\}$$

If R is the above record type and $r : R$, then we denote access the component of the record as follows.

$$\begin{array}{c} r[\text{field}_1] : A_1\\ r[\text{field}_2] : A_2\\ \vdots\\ r[\text{field}_n] : A_n\end{array}$$

To construct a record value given values $a_1 : A_1, ..., a_n : A_n$, we again use a curly brace notation.

$$\left\{\begin{array}{l}\text{field}_1 := a_1\\ \text{field}_2 := a_2\\ \vdots\\ \text{field}_n := a_n\end{array}\right\} : \left\{\begin{array}{l}\text{field}_1 : A_1\\ \text{field}_2 : A_2\\ \vdots\\ \text{field}_n : A_n\end{array}\right\}$$

2.2 Functors

A functor $\mathcal{F}$ is a type parameterized by a free type variable, such that whenever A is a type then $\mathcal{F}(A)$ is a type. Given a function $f : A \to B$, is it often possible to define a new function $\mathcal{F}f : \mathcal{F}A \to \mathcal{F}B$ such that for all $f : A \to B$ and $g : B \to C$ we have

$$\begin{array}{rcl}\mathcal{F}\text{id}_A & = & \text{id}_{\mathcal{F}A}\\ \mathcal{F}(f \circ g) & = & \mathcal{F}f \circ \mathcal{F}g\end{array}$$

When this happens we call such a functor a *covariant functor*. In this document all our functors will be covariant functors, and we will simply call them *functor*s.

2.2.1 Option Functor

By way of a useful example, we define S to be the option functor, also known as the maybe functor,

$$\begin{array}{rcl}\text{S}A & := & \mathbb{1} + A\\ \text{S}f(\sigma^{\mathbf{L}}_{\mathbb{1},A}\langle\rangle) & := & \sigma^{\mathbf{L}}_{\mathbb{1},B}\langle\rangle\\ \text{S}f(\sigma^{\mathbf{R}}_{\mathbb{1},A}(a)) & := & \sigma^{\mathbf{R}}_{\mathbb{1},B}(f(a))\end{array}$$

where $f : A \to B$.

We define special notation for values of "optional" types SA,

$$\begin{array}{rcl}\emptyset^{\text{S}}_A & := & \sigma^{\mathbf{L}}_{\mathbb{1},A}\langle\rangle : \text{S}A\\ \eta^S_A(a) & := & \sigma^{\mathbf{R}}_{\mathbb{1},A}(a) : \text{S}A\end{array}$$

where $a : A$.

This notation is designed to coincide with the monadic notation that we will define in Section 2.3.4.

2.2.2 List Functors

Given a type A, we recursively define the list functor A^* and the non-empty list functor A^+,

$$\begin{aligned} A^* &:= \mathrm{S}A^+ \\ A^+ &:= A \times A^* \\ f^*(\emptyset^{\mathrm{S}}_{A^+}) &:= \emptyset^{\mathrm{S}}_{B^+} \\ f^*(\eta^{\mathrm{S}}_{A^+}(l)) &:= \eta^{\mathrm{S}}_{B^+}(f^+(l)) \\ f^+\langle a, l\rangle &:= \langle f(a), f^*(l)\rangle \end{aligned}$$

where $f : A \to B$.

We leave implicit the fact that these are inductive types and recursive definitions over them need to be checked that they are well-founded. Suffice to say that the definitions in this section are all well-defined.

Given a list $l : A^*$ or a non-empty list $l : A^+$, we define $|l| : \mathbb{N}$ to be its length.

$$\begin{aligned} |\emptyset^{\mathrm{S}}_{A^+}| &:= 0 \\ |\eta^{\mathrm{S}}_{A^+}(l)| &:= |l| \\ |\langle a, l\rangle| &:= 1 + |l| \end{aligned}$$

To retrieve an element we define lookup functions for lists and non-empty lists. Given a natural number $n : \mathbb{N}$ and either list $l : A^*$ or a non-empty list $l : A^+$, in both cases we define $l[n] : \mathrm{S}A$ to lookup the nth value in l.

$$\begin{aligned} (\emptyset^{\mathrm{S}}_{A^+})[n] &:= \emptyset^{\mathrm{S}}_{A} \\ (\eta^{\mathrm{S}}_{A^+}(l))[n] &:= l[n] \\ \langle a, l\rangle[0] &:= \eta^{\mathrm{S}}_{A}(a) \\ \langle a, l\rangle[1+n] &:= l[n] \end{aligned}$$

Naturally, the lookup returns $\emptyset$ when the index goes beyond the end of the list.

Lemma 2.1. *For all $n : \mathbb{N}$ and $l : A^*$ or $l : A^+$, $l[n] = \emptyset$ if and only if $|l| \leq n$.*

Given a list $l : A^*$ (or a non-empty list $l : A^+$), we define $\mathrm{indexed}(l) : (\mathbb{N} \times A)^*$ (and $\mathrm{indexed}(l) : (\mathbb{N} \times A)^+$ respectively) as a list of elements paired with its index,

$$\mathrm{indexed}(l) := \mathrm{indexedRec}(0, l)$$

where

$$\begin{aligned} \mathrm{indexedRec}(i, \emptyset^{\mathrm{S}}_{A^+}) &:= \emptyset^{\mathrm{S}}_{(\mathbb{N}\times A)^+} \\ \mathrm{indexedRec}(i, \eta^{\mathrm{S}}_{A^+}(l)) &:= \eta^{\mathrm{S}}_{(\mathbb{N}\times A)^+}(\mathrm{indexedRec}(i, l)) \\ \mathrm{indexedRec}(i, \langle a, l\rangle) &:= \langle\langle i, a\rangle, \mathrm{indexedRec}(1+i, l)\rangle. \end{aligned}$$

Lemma 2.2. *For all $i : \mathbb{N}$ and $l : A^*$ or $l : A^+$, $\mathrm{indexed}(l)[i] = \mathrm{S}(\lambda a.\langle i, a\rangle)(l[i])$.*

The fold operation on a list is a most general way of consuming a list. Given a type A with a monoid $\langle \odot, e\rangle$ over that type, we define $\mathrm{fold}^{\langle\odot,e\rangle}_A(l)$ for both lists $l : A^*$ and non-empty lists $l : A^+$.

$$\begin{aligned} \mathrm{fold}^{\langle\odot,e\rangle}_A(\emptyset^{\mathrm{S}}_{A^+}) &:= e \\ \mathrm{fold}^{\langle\odot,e\rangle}_A(\eta^{\mathrm{S}}_{A^+}(l)) &:= \mathrm{fold}^{\langle\odot,e\rangle}_A(l) \\ \mathrm{fold}^{\langle\odot,e\rangle}_A\langle a, l\rangle &:= a \odot \mathrm{fold}^{\langle\odot,e\rangle}_A(l) \end{aligned}$$

Often we will write $\mathrm{fold}^{\langle\odot,e\rangle}_A(l)$ as simply $\mathrm{fold}^{\odot}(l)$ since usually both the type and the unit for a monoid is can be inferred from just its binary operation.

For lists, we provide special notations for its two effective constructors called *nil*, $\epsilon_A : A^*$, and *cons*, $a \blacktriangleleft l : A^*$

$$\begin{aligned} \epsilon_A &:= \emptyset^{\mathrm{S}}_{A^+} \\ a \blacktriangleleft l &:= \eta^{\mathrm{S}}_{A^+}\langle a, l\rangle \end{aligned}$$

where $a : A$ and $l : A^*$. As a consequence the following equations hold.

$$\begin{aligned} \text{fold}_A^{\langle \odot, e \rangle}(\epsilon_A) &= e \\ \text{fold}_A^{\langle \odot, e \rangle}(a_0 \blacktriangleleft a_1 \blacktriangleleft \ldots \blacktriangleleft a_n \blacktriangleleft \epsilon_A) &= a_0 \odot a_1 \odot \ldots \odot a_n \end{aligned}$$

For example, given two lists $l_1, l_2 : A^*$, we define the append operation $l_1 \cdot l_2 : A^*$ using nil and cons

$$\begin{aligned} \epsilon \cdot l &:= l \\ (a \blacktriangleleft l_1) \cdot l_2 &:= a \blacktriangleleft (l_1 \cdot l_2) \end{aligned}$$

The append operation together with nil, $\langle \cdot, \epsilon \rangle$, forms a monoid over A^*. This allows us to define the concatenation function $\mu_A^* : A^{**} \to A^*$

$$\mu_A^*(l) := \text{fold}_{A^*}^{\langle \cdot, \epsilon \rangle}(l)$$

Now it is only natural to define a function that generates a list with one element, $\eta_A^* : A \to A^*$.

$$\eta_A^*(a) := a \blacktriangleleft \epsilon$$

We write replication of a list, $l : A^*$ as exponentiation:

$$\begin{aligned} l^0 &:= \epsilon \\ l^{1+n} &:= l \cdot l^n \end{aligned}$$

We also define quantifier notation for elements of lists. Given a list $l : A^*$, and a predicate P over A, we define

$$\begin{aligned} \forall a \in l.\, P(a) &:= \text{fold}^{\wedge}(P^*(l)) \\ \exists a \in l.\, P(a) &:= \text{fold}^{\vee}(P^*(l)) \end{aligned}$$

and similarly for a non-empty list $l : A^+$:

$$\begin{aligned} \forall a \in l.\, P(a) &:= \text{fold}^{\wedge}(P^+(l)) \\ \exists a \in l.\, P(a) &:= \text{fold}^{\vee}(P^+(l)) \end{aligned}$$

2.2.3 Buffers

A buffer is defined as a bounded list. We recursively define a buffer upto (but excluding) 2^n as

$$\begin{aligned} A^{<2} &:= \mathrm{S}A \\ A^{<4} &:= \mathrm{S}(A^2) \times A^{<2} \\ &\vdots \\ A^{<2^{1+n}} &:= \mathrm{S}(A^{2^n}) \times A^{<2^n} \\ &\vdots \end{aligned}$$

Notice that the type for buffers is isomorphic to a finite geometric series of tuples:

$$A^{<2^n} \asymp 1 + A + (A \times A) + (A \times A \times A) + \ldots + \underbrace{(A \times \ldots \times A)}_{2^n - 1 \text{ times}}$$

2.3 Monads

A monad, $\mathcal{M}$, is a functor that comes with two functions

$$\begin{aligned} \eta_A^{\mathcal{M}} &: A \to \mathcal{M}\, A \\ \mu_A^{\mathcal{M}} &: \mathcal{M}\, \mathcal{M}\, A \to \mathcal{M}\, A \end{aligned}$$

that are both natural transformations, meaning for all $f : A \to B$,

$$\begin{aligned} \mathcal{M}\, f \circ \eta_A^{\mathcal{M}} &= \eta_B^{\mathcal{M}} \circ f \\ \mathcal{M}\, f \circ \mu_A^{\mathcal{M}} &= \mu_B^{\mathcal{M}} \circ \mathcal{M}\, \mathcal{M}\, f \end{aligned}$$

The $\eta_A^{\mathcal{M}}$ and $\mu_A^{\mathcal{M}}$ functions are required to satisfy certain coherence laws. These monad laws are best presented using Kleisli composition.

2.3.1 Kleisli Morphisms

Functions from A to B that produce side-effects can often be represented by Kleisli morphisms, which are (pure) functions $A \to \mathcal{M}\ B$, where $\mathcal{M}$ is a monad that captures the particular side-effects of the function in the result. A function $f : A \to \mathcal{M}\ B$ is called a *Kleisli morphism* from A to B. For Kleisli morphisms $f : A \to \mathcal{M}\ B$ and $g : B \to \mathcal{M}\ C$ we define the *Kleisli composition* of them as $g \overset{\mathcal{M}}{\leftarrowtail} f : A \to \mathcal{M}\ C$ where

$$g \overset{\mathcal{M}}{\leftarrowtail} f := \mu^{\mathcal{M}} \circ \mathcal{M}\ g \circ f$$

We will usually omit the annotation.

The monad laws can be presented in terms of Kleisli composition. For all $f : A \to \mathcal{M}\ B$, $g : B \to \mathcal{M}\ C$, and $h : C \to \mathcal{M}\ D$, we require that Kleisli composition satisfy the laws of composition with $\eta^{\mathcal{M}}$ as its identity:

$$\begin{aligned} f \leftarrowtail \eta_A^{\mathcal{M}} &= f \\ \eta_B^{\mathcal{M}} \leftarrowtail f &= f \\ (h \leftarrowtail g) \leftarrowtail f &= h \leftarrowtail (g \leftarrowtail f) \end{aligned}$$

2.3.2 Cartesian Strength

In addition to Kleisli composition we define a series of helper functions for manipulating products.

$$\begin{aligned} \beta_{A,B}^{\mathcal{M}} &: A \times \mathcal{M}\ B \to \mathcal{M}(A \times B) \\ \beta^{\mathcal{M}}\langle a, b\rangle &:= \mathcal{M}(\lambda x.\ \langle a, x\rangle)(b) \end{aligned}$$

$$\begin{aligned} \bar{\beta}_{A,B}^{\mathcal{M}} &: \mathcal{M}\ A \times B \to \mathcal{M}(A \times B) \\ \bar{\beta}^{\mathcal{M}}\langle a, b\rangle &:= \mathcal{M}(\lambda x.\ \langle x, b\rangle)(a) \end{aligned}$$

$$\begin{aligned} \phi_{A,B}^{\mathcal{M}} &: \mathcal{M}\ A \times \mathcal{M}\ B \to \mathcal{M}(A \times B) \\ \phi^{\mathcal{M}} &:= \beta^{\mathcal{M}} \leftarrowtail \bar{\beta}^{\mathcal{M}} \end{aligned}$$

$$\begin{aligned} \bar{\phi}_{A,B}^{\mathcal{M}} &: \mathcal{M}\ A \times \mathcal{M}\ B \to \mathcal{M}(A \times B) \\ \bar{\phi}^{\mathcal{M}} &:= \bar{\beta}^{\mathcal{M}} \leftarrowtail \beta^{\mathcal{M}} \end{aligned}$$

The operations $\phi_{A,B}^{\mathcal{M}}$ and $\bar{\phi}_{A,B}^{\mathcal{M}}$ are similar, but differ in the order that effects are applied in. Roughly speaking, $\phi_{A,B}^{\mathcal{M}}$ applies the effects of the first component first, while $\bar{\phi}_{A,B}^{\mathcal{M}}$ applies the effects of the second component first. For some monads, the order of the effects is immaterial and $\phi_{A,B}^{\mathcal{M}} = \bar{\phi}_{A,B}^{\mathcal{M}}$. We call such monads *commutative monads*.

It is always the case that

$$\phi_{A,A}^{\mathcal{M}} \circ \Delta_{\mathcal{M}\ A} = \bar{\phi}_{A,A}^{\mathcal{M}} \circ \Delta_{\mathcal{M}\ A} : \mathcal{M}\ A \to \mathcal{M}(A^2)$$

holds, even for non-commutative monads. In this case, the effect specified by the input is duplicated. Compare this with $\mathcal{M}\ \Delta_A : \mathcal{M}\ A \to \mathcal{M}(A^2)$ where the contents of type A are duplicated, but not the effect itself. When we have $\mathcal{M}\ \Delta_A = \phi_{A,A}^{\mathcal{M}} \circ \Delta_{\mathcal{M}\ A}$ (equiv. $\mathcal{M}\ \Delta_A = \bar{\phi}_{A,A}^{\mathcal{M}} \circ \Delta_{\mathcal{M}\ A}$), we say that $\mathcal{M}$ is an *idempotent monad*.[2.1] For idempotent monads, the same effect is occurring two or more times in a row is equivalent to it occurring once.

2.3.3 Identity Monad

The most trivial monad is the identity monad, Id, where $\mathrm{Id}\ A := A$ and $\mathrm{Id}\ f := f$. The natural transformations η_A^{Id} and μ_A^{Id} are both the identity function. The identity monad captures no side-effects and it is commutative and idempotent.

2.1. Beware that we are using the definition of idempotent monad from King and Wadler [?], as opposed to the traditional categorical definition of an idempotent monad.

2.3.4 Monad Zero

Some monads have a universal *zero* value

$$\emptyset_A^{\mathcal{M}} : \mathcal{M}\, A$$

where for all $f : A \to B$,

$$\mathcal{M} f(\emptyset_A^{\mathcal{M}}) = \emptyset_B^{\mathcal{M}}$$

This zero value denotes a side-effect that captures the notion of a failed or aborted computation or some kind of empty result.

The laws for these monads with zero are, again, best expressed using Kleisli morphisms. At the risk of some notational confusion, we define $\varnothing_{A,B}^{\mathcal{M}} : A \to \mathcal{M}\, B$ as a *zero morphism*.

$$\varnothing_{A,B}^{\mathcal{M}}(a) := \emptyset_B^{\mathcal{M}}$$

Zero morphisms are required to be an absorbing element for Kleisli composition. For all $f : A \to \mathcal{M}\, B$ and $g : B \to \mathcal{M}\, C$ we require that

$$\begin{aligned} g \leftarrowtail \varnothing_{A,B}^{\mathcal{M}} &= \varnothing_{A,C}^{\mathcal{M}} \\ \varnothing_{B,C}^{\mathcal{M}} \leftarrowtail f &= \varnothing_{A,C}^{\mathcal{M}} \end{aligned}$$

(Note: In Haskell, the `mzero` value typically is only required to satisfy the first equation; however, we require monads with zero to satisfy both laws.)

2.3.4.1 Option Monad

The functor S forms a monad with the following operations:

$$\begin{aligned} \eta_A^{\mathrm{S}}(a) &:= \sigma^{\mathbf{R}}(a) \\ \mu_A^{S}(\sigma^{\mathbf{L}}\langle\rangle) &:= \sigma^{\mathbf{L}}\langle\rangle \\ \mu_A^{S}(\sigma^{\mathbf{R}}(x)) &:= x \end{aligned}$$

The option monad is commutative and idempotent. The option monad has a zero:

$$\emptyset_A^{\mathrm{S}} := \sigma^{\mathbf{L}}\langle\rangle$$

There is a natural transformation from the option monad into any monad with zero, $\iota_{\mathrm{S},A}^{\mathcal{M}} : \mathrm{S}A \to \mathcal{M}\, A$:

$$\begin{aligned} \iota_{\mathrm{S},A}^{\mathcal{M}}(\emptyset_A^{\mathrm{S}}) &:= \emptyset_A^{\mathcal{M}} \\ \iota_{\mathrm{S},A}^{\mathcal{M}}(\eta_A^{\mathrm{S}}(a)) &:= \eta_A^{\mathcal{M}}(a) \end{aligned}$$

Lemma 2.3. *For all $f : A \to B$,*

$$\iota_{\mathrm{S},B}^{\mathcal{M}} \circ \mathrm{S} f = \mathcal{M}\, f \circ \iota_{\mathrm{S},A}^{\mathcal{M}}$$

Also

$$\iota_{\mathrm{S},A}^{\mathcal{M}} \circ \mu_A^{\mathrm{S}} = \mu_A^{\mathcal{M}} \circ \iota_{\mathrm{S},\mathcal{M}A}^{\mathcal{M}} \circ \mathrm{S}\iota_{\mathrm{S},A}^{\mathcal{M}}\ (= \mu_A^{\mathcal{M}} \circ \mathcal{M}\, \iota_{\mathrm{S},A}^{\mathcal{M}} \circ \iota_{\mathrm{S},\mathrm{S}A}^{\mathcal{M}})$$

2.4 Multi-bit Words

By repeatedly taking products of the bit type we can build the types 2^{2^n} which denote 2^n-bit words. We choose to represent values in big endian format, meaning that given a pair representing the low and high bits of a value, the most significant bits are stored in the first half. Given a value $a : 2^n$, where n is a power of two, we recursively define $\lceil a \rceil_n : \mathbb{N}$ to be the number that a represents:

$$\begin{aligned} \lceil 0_2 \rceil_1 &:= 0 \\ \lceil 1_2 \rceil_1 &:= 1 \\ \lceil \langle a, b \rangle \rceil_{2n} &:= \lceil a \rceil_n\, 2^n + \lceil b \rceil_n \end{aligned}$$

We also make use of the following variation of this value interpretation function.

$$\lceil\langle a, b\rangle\rceil_{n,m} := \lceil a\rceil_n\, 2^m + \lceil b\rceil_m$$

These value interpretation functions are all injective (one-to-one) and we can choose a left inverse. Given $m : \mathbb{N}$, we implicitly define $\lfloor m\rfloor_n : 2^n$ such that $\lceil\lfloor m\rfloor_n\rceil_n \equiv m \pmod{2^n}$. We have chosen $\lfloor m\rfloor_n$ so that it represents m modulo 2^n.

We can equip the type 2^n with addition and multiplication operations, so that $\lfloor\cdot\rfloor_n$ becomes a semiring homomorphism. Given $a : 2^n$ and $b : 2^n$, we define $a \lfloor+\rfloor_n b : 2^n$ and $a \lfloor\times\rfloor_n b : 2^n$ such that

$$\begin{aligned} \lfloor m_1\rfloor_n \lfloor+\rfloor_n \lfloor m_2\rfloor_n &= \lfloor m_1 + m_2\rfloor_n \\ \lfloor m_1\rfloor_n \lfloor\times\rfloor_n \lfloor m_2\rfloor_n &= \lfloor m_1\, m_2\rfloor_n \end{aligned}$$

We write $\mathtt{0}_{2^4}, \mathtt{1}_{2^4}, ..., \mathtt{f}_{2^4}$ to denote the 16 values of 2^4 that represent their respective hexadecimal values. Similarly, we write $\mathtt{00}_{2^8}, \mathtt{01}_{2^8}, ..., \mathtt{ff}_{2^8}$ to denote the 256 values of 2^8, and so forth. It is worth observing that for hexadecimal digits $\mathtt{x}$ and $\mathtt{y}$, we have $\mathtt{xy}_{2^8} = \langle \mathtt{x}_{2^4}, \mathtt{y}_{2^4}\rangle$.

2.4.1 Byte Strings

The type $(2^8)^*$ is known as byte strings. We will write byte string values as sequences of hexadecimal digits surrounded by square brackets, e.g. $\mathtt{[cafe]}_{2^8}$ denotes $\mathtt{ca}_{2^8} \blacktriangleleft \mathtt{fe}_{2^8} \blacktriangleleft \epsilon_{2^8}$ (whereas $\mathtt{cafe}_{2^{16}}$ denotes $\langle \mathtt{ca}_{2^8}, \mathtt{fe}_{2^8}\rangle$). For all these values, we may omit the subscript when the interpretation is clear from the context.

Words larger than a byte are commonly encoded as byte strings in either big endian or little endian order. We define $\mathrm{BE}_n : 2^n \to (2^8)^*$ and $\mathrm{LE}_n : 2^n \to (2^8)^*$ as big endian and little endian encodings of words respectively for $n \geq 8$.

$$\begin{aligned} \mathrm{LE}_8(a) &:= \eta^*(a) \\ \mathrm{LE}_{2n}\langle a_1, a_2\rangle &:= \mathrm{LE}_n(a_2)\cdot\mathrm{LE}_n(a_1) && \text{(when } 8 \leq n\text{)} \\ \mathrm{BE}_8(a) &:= \eta^*(a) \\ \mathrm{BE}_{2n}\langle a_1, a_2\rangle &:= \mathrm{BE}_n(a_1)\cdot\mathrm{BE}_n(a_2) && \text{(when } 8 \leq n\text{)} \end{aligned}$$

2.4.2 Bit Strings

The type 2^* is known as bit strings. We will write bit string values as sequences of binary digits surrounded by square brackets, e.g. $\mathtt{[0110]}_2$ denotes $\mathtt{0}_2 \blacktriangleleft \mathtt{1}_2 \blacktriangleleft \mathtt{1}_2 \blacktriangleleft \mathtt{0}_2 \blacktriangleleft \epsilon_2$. Again, we may omit the subscript when the interpretation is clear from context.

Chapter 3
Core Simplicity

Simplicity is a typed functional programming language based on Gentzen's sequent calculus [?]. The core language consists of nine combinators for forming expressions. These nine combinators capture the computational power of Simplicity. In later chapters other combinators will extend this core language and provide other effects to handle input and access the transaction that provides context for the Simplicity program.

3.1 Types

This section introduces the abstract syntax and semantics types available in Simplicity. Simplicity uses a particular subset of the simple type theory we developed in Chapter 2.

3.1.1 Abstract Syntax

Simplicity has only three kinds of types:

- The unit type, $\mathbb{1}$.
- The sum of two Simplicity types, $A + B$.
- The product of two Simplicity types, $A \times B$.

Simplicity has neither function types nor recursive types. Every type in Simplicity can only contain a finite number of values. For example, the type 2, which is $\mathbb{1} + \mathbb{1}$, has exactly two values, namely $\sigma^{\mathbf{L}}_{\mathbb{1},\mathbb{1}}\langle\rangle$ and $\sigma^{\mathbf{R}}_{\mathbb{1},\mathbb{1}}\langle\rangle$. The type $(\mathbb{1} + \mathbb{1}) \times (\mathbb{1} + \mathbb{1})$ has exactly four values. As you can see, the number of values that a type contains can be easily calculated by interpreting the type as an arithmetic expression. Be aware that types are not arithmetic expressions. For example, the types $(\mathbb{1} + \mathbb{1}) + (\mathbb{1} + \mathbb{1})$ and $(\mathbb{1} + \mathbb{1}) \times (\mathbb{1} + \mathbb{1})$ are distinct and not interchangeable.

3.1.2 Formal Syntax

Formally we define the abstract syntax of types as an inductive type in Coq:

```
Inductive Ty : Set :=
| Unit : Ty
| Sum  : Ty -> Ty -> Ty
| Prod : Ty -> Ty -> Ty.
```

3.1.3 Formal Semantics

Formally we define the denotational semantics of Simplicity types as a function from syntax to Coq types:

```
Fixpoint tySem (X : Ty) : Set :=
match X with
| Unit => Datatypes.unit
| Sum A B => tySem A + tySem B
| Prod A B => tySem A * tySem B
end.
```

3.2 Terms

Simplicity programs are composed of terms that denote functions between types. Every Simplicity term is associated with an input type and an output type and we write a type annotated term as $t : A \vdash B$ where t is the term, A is the input type and B is the output type. We write $[\![t]\!] : A \to B$ for the function that the term t denotes.

Core Simplicity has nine combinators for forming well-typed terms.

3.2.1 Identity

$$\frac{}{\mathsf{iden}_A : A \vdash A}$$

$$[\![\mathsf{iden}_A]\!] := \lambda a.\, a$$

For every Simplicity type A, we have an identity term that denotes the identity function for that type.

We can also write the semantics in application style as

$$[\![\mathsf{iden}_A]\!](a) := a$$

which is just a different way of writing the same definition. However, please note that the a argument is an argument of the function denoted by iden_A and is not an argument to the Simplicity term itself.

3.2.2 Composition

$$\frac{s : A \vdash B \quad t : B \vdash C}{\mathsf{comp}_{A,B,C}\, s\, t : A \vdash C}$$

$$[\![\mathsf{comp}_{A,B,C}\, s\, t]\!] := [\![t]\!] \circ [\![s]\!]$$

The composition combinator functionally composes its two arguments, s and t, when the output type of s matches the input type of t.

3.2.3 Constant Unit

$$\frac{}{\mathsf{unit}_A : A \vdash \mathbb{1}}$$

$$[\![\mathsf{unit}_A]\!] := \lambda a.\, \langle\rangle$$

The unit term denotes the constant function that always returns $\langle\rangle$, the unique value of the unit type. The function's argument is ignored and we have a constant unit term for every type of input.

We can also write semantics in application style as

$$[\![\mathsf{unit}_A]\!](a) := \langle\rangle.$$

We will use application style when writing the definition of the remaining combinators, trusting that the reader will be mindful of the distinction between arguments of the combinators versus the argument of the function that the Simplicity term denotes.

3.2.4 Left Injection

$$\frac{t : A \vdash B}{\mathsf{injl}_{A,B,C}\, t : A \vdash B + C}$$

$$[\![\mathsf{injl}_{A,B,C}\, t]\!](a) := \sigma^{\mathbf{L}}([\![t]\!](a))$$

The left injection combinator composes a left-tag with its argument t.

3.2.5 Right Injection

$$\frac{t : A \vdash C}{\mathsf{injr}_{A,B,C}\, t : A \vdash B + C}$$

$$[\![\mathsf{injr}_{A,B,C}\, t]\!](a) := \sigma^{\mathbf{R}}([\![t]\!](a))$$

The right injection combinator composes a right-tag with its argument t.

3.2.6 Case

$$\frac{s : A \times C \vdash D \qquad t : B \times C \vdash D}{\mathsf{case}_{A,B,C,D}\, s\, t : (A + B) \times C \vdash D}$$

$$\begin{aligned} [\![\mathsf{case}_{A,B,C,D}\, s\, t]\!]\langle \sigma^{\mathbf{L}}(a), c\rangle &:= [\![s]\!]\langle a, c\rangle \\ [\![\mathsf{case}_{A,B,C,D}\, s\, t]\!]\langle \sigma^{\mathbf{R}}(b), c\rangle &:= [\![t]\!]\langle b, c\rangle \end{aligned}$$

The case combinator is Simplicity's only branching operation. Given a pair of values with the first component being a sum type, this combinator evaluates either its s or t argument, depending on which tag the first component has, on the pair of inputs.

3.2.7 Pair

$$\frac{s : A \vdash B \qquad t : A \vdash C}{\mathsf{pair}_{A,B,C}\, s\, t : A \vdash B \times C}$$

$$[\![\mathsf{pair}_{A,B,C}\, s\, t]\!](a) := \langle [\![s]\!](a), [\![t]\!](a)\rangle$$

The pair combinator evaluates both its arguments, s and t, on the same input and returns the pair of the two results.

3.2.8 Take

$$\frac{t : A \vdash C}{\mathsf{take}_{A,B,C}\, t : A \times B \vdash C}$$

$$[\![\mathsf{take}_{A,B,C}\, t]\!]\langle a, b\rangle := [\![t]\!](a)$$

The take combinator denotes a function on pairs that passes its first component to t and ignores its second component.

3.2.9 Drop

$$\frac{t : B \vdash C}{\mathsf{drop}_{A,B,C}\, t : A \times B \vdash C}$$

$$[\![\mathsf{drop}_{A,B,C}\, t]\!]\langle a, b\rangle := [\![t]\!](b)$$

The drop combinator denotes a function on pairs that passes its second component to t and ignores its first component.

3.2.10 Formal Syntax

We define the formal syntax of well-typed core Simplicity terms as an inductive family in Coq:

```
Inductive Term : Ty -> Ty -> Set :=
| iden : forall {A}, Term A A
| comp : forall {A B C}, Term A B -> Term B C -> Term A C
| unit : forall {A}, Term A Unit
| injl : forall {A B C}, Term A B -> Term A (Sum B C)
| injr : forall {A B C}, Term A C -> Term A (Sum B C)
| case : forall {A B C D},
    Term (Prod A C) D -> Term (Prod B C) D -> Term (Prod (Sum A B) C) D
| pair : forall {A B C}, Term A B -> Term A C -> Term A (Prod B C)
```

```
| take : forall {A B C}, Term A C -> Term (Prod A B) C
| drop : forall {A B C}, Term B C -> Term (Prod A B) C.
```

3.2.11 Formal Semantics

The formal semantics for core Simplicity in Coq recursively interprets each term as a function between the formal semantics of its associated types:

```
Fixpoint eval {A B} (x : Term A B) : tySem A -> tySem B :=
match x in Term A B return tySem A -> tySem B with
| iden => fun a => a
| comp s t => fun a => eval t (eval s a)
| unit => fun _ => tt
| injl t => fun a => inl (eval t a)
| injr t => fun a => inr (eval t a)
| case s t => fun p => let (ab, c) := p in
    match ab with
    | inl a => eval s (a, c)
    | inr b => eval t (b, c)
    end
| pair s t => fun a => (eval s a, eval t a)
| take t => fun ab => eval t (fst ab)
| drop t => fun ab => eval t (snd ab)
end.
```

3.3 Example Simplicity

Simplicity is not meant to be a language to directly write programs in. It is intended to be a backend language that some other language (or languages) is complied or translated to. However, one can program directly in Simplicity just as one can write programs directly in an assembly language.

Because the core Simplicity language may seem meager, it is worthwhile to see how one can build up sophisticated functions in it.

3.3.1 Bit Operations

For the bit type, 2, we can define core Simplicity terms that represent the two constant functions that return this type:

$$\begin{aligned} \mathsf{false}_A &:= \mathsf{injl}_{A,\mathbb{1},\mathbb{1}}\,\mathsf{unit} : A \vdash 2 \\ \mathsf{true}_A &:= \mathsf{injr}_{A,\mathbb{1},\mathbb{1}}\,\mathsf{unit} : A \vdash 2 \end{aligned}$$

From these definitions, we can prove that false and true have the following semantics.

$$\begin{aligned} [\![\mathsf{false}]\!](a) &= 0_2 \\ [\![\mathsf{true}]\!](a) &= 1_2 \end{aligned}$$

Next, we define a condition combinator to branch based on the value of a bit using case and drop. The first argument is the "then" clause and the second argument is the "else" clause.

$$\frac{s : A \vdash B \qquad t : A \vdash B}{\mathsf{cond}_{A,B}\,s\,t := \mathsf{case}_{\mathbb{1},\mathbb{1},A,B}\,(\mathsf{drop}\,t)\,(\mathsf{drop}\,s) : 2 \times A \vdash B}$$

We can prove that cond has the following semantics.

$$\begin{aligned} [\![\mathsf{cond}\,s\,t]\!]\langle 1_2, a\rangle &= [\![s]\!](a) \\ [\![\mathsf{cond}\,s\,t]\!]\langle 0_2, a\rangle &= [\![t]\!](a) \end{aligned}$$

With these fundamental operations for bits in hand, we can define standard Boolean connectives:

$$\frac{t : A \vdash 2}{\mathsf{not}_A\, t := \mathsf{comp}_{A,2\times\mathbb{1},2}\,(\mathsf{pair}\, t\, \mathsf{unit})\,(\mathsf{cond}\,\mathsf{false}\,\mathsf{true}) : A \vdash 2}$$

$$\frac{s : A \vdash 2 \qquad t : A \vdash 2}{\mathsf{and}_A\, s\, t := \mathsf{comp}_{A,2\times A,2}\,(\mathsf{pair}\, s\,\mathsf{iden})\,(\mathsf{cond}\, t\,\mathsf{false}) : A \vdash 2}$$

$$\frac{s : A \vdash 2 \qquad t : A \vdash 2}{\mathsf{or}_A\, s\, t := \mathsf{comp}_{A,2\times A,2}\,(\mathsf{pair}\, s\,\mathsf{iden})\,(\mathsf{cond}\,\mathsf{true}\, t) : A \vdash 2}$$

We use combinators to define and and or in order to give them short-circuit evaluation behaviour. Short-circuit evaluation useful because if we know the second branch does not need to be evaluated, the source code for it can be pruned at redemption time (see Section 4.3.2.1). If instead we directly defined the Boolean functions with types and-func: $2 \times 2 \vdash 2$ and or-func: $2 \times 2 \vdash 2$, then the two arguments to and-func and or-func would both be fully evaluated under strict semantics (see Section 4.1). For the not combinator, this is less of an issue, but we define it in combinator form to be consistent.

3.3.2 Simplicity Notation

In the previous section, we were relatively detailed with the annotations given to the definitions. Going forward we will be a bit more lax in the presentation. We will also begin using some notation to abbreviate terms.

$$\begin{aligned} s \vartriangle t &:= \mathsf{pair}\, s\, t \\ s; t &:= \mathsf{comp}\, s\, t \end{aligned}$$

with the $\vartriangle$ operator having higher precedence than the ; operator.

Composition of sequences of drop and take with iden is a very common way of picking data from a nested tuple input. To make this more concise we will use the following notation.

$$\begin{aligned} \mathsf{H} &:= \mathsf{iden} \\ \mathsf{O}s &:= \mathsf{take}\, s \\ \mathsf{I}s &:= \mathsf{drop}\, s \end{aligned}$$

where s is a string of I's and O's that ends with H.

3.3.3 Generic Equality

With our notion of a bit in hand, we can build Simplicity predicates of type $A \vdash 2$. For example, for any Simplicity type A we can build a Simplicity predicate $\mathsf{eq}_A : A \times A \vdash 2$, that decides if two values of the same type are equal or not.

$$\begin{aligned} \mathsf{eq}_{\mathbb{1}} &:= \mathsf{true} \\ \mathsf{eq}_{A+B} &:= \mathsf{case}\,(\mathsf{IH} \vartriangle \mathsf{OH}; \mathsf{case}\,\mathsf{eq}_A\,\mathsf{false})\,(\mathsf{IH} \vartriangle \mathsf{OH}; \mathsf{case}\,\mathsf{false}\,\mathsf{eq}_B) \\ \mathsf{eq}_{A\times B} &:= (\mathsf{OOH} \vartriangle \mathsf{IOH}; \mathsf{eq}_A) \vartriangle (\mathsf{OIH} \vartriangle \mathsf{IIH}); \mathsf{cond}\,\mathsf{eq}_B\,\mathsf{false} \end{aligned}$$

Theorem 3.1. *Given any Simplicity type A, for values $a_0, a_1 : A$,*

$$[\![\mathsf{eq}_A]\!]\langle a_0, a_1\rangle = 1_2 \text{ if and only if } a_0 = a_1.$$

3.3.4 Arithmetic

Using techniques familiar from digital logic, we can build an adders and full adders from our Boolean operations defined in Section 3.3.1. We begin with definitions of the single bit full adder.

$$\begin{aligned}&\mathsf{full\text{-}add}_1 : 2 \times (2 \times 2) \vdash 2^2\\ &\mathsf{full\text{-}add}_1 \;:=\; \mathsf{maj}_1 \vartriangle \mathsf{xor3}_1\end{aligned}$$

where

$$\begin{aligned}&\mathsf{maj}_1 : 2 \times (2 \times 2) \vdash 2\\ &\mathsf{maj}_1 \;:=\; \mathsf{cond}\,(\mathsf{cond}\,\mathsf{true}\,\mathsf{iden})\,(\mathsf{cond}\,\mathsf{iden}\,\mathsf{false})\end{aligned}$$

and

$$\begin{aligned}&\mathsf{xor3}_1 : 2 \times (2 \times 2) \vdash 2\\ &\mathsf{xor3}_1 \;:=\; \mathsf{cond}\,(\mathsf{cond}\,\mathsf{iden}\,(\mathsf{not}\,\mathsf{iden}))\,(\mathsf{cond}\,(\mathsf{not}\,\mathsf{iden})\,\mathsf{iden})\end{aligned}$$

The full adder meets the following specification.

$$\lceil [\![\mathsf{full\text{-}add}_1]\!] \langle c, \langle a, b\rangle\rangle \rceil_2 \;=\; \lceil a \rceil_1 + \lceil b \rceil_1 + \lceil c \rceil_1$$

It is easy to exhaustively check the above equations because there are only eight inputs to consider. We will illustrate this for a single case where $a = 1_2$, $b = 0_2$, and $c = 1_2$.

$$\begin{aligned}
\lceil [\![\mathsf{full\text{-}add}_1]\!] \langle 1_2, \langle 1_2, 0_2\rangle\rangle \rceil_2 &= \lceil [\![\mathsf{maj}_1 \vartriangle \mathsf{xor3}_1]\!] \langle 1_2, \langle 1_2, 0_2\rangle\rangle \rceil_2\\
&= \lceil \langle [\![\mathsf{maj}_1]\!] \langle 1_2, \langle 1_2, 0_2\rangle\rangle, [\![\mathsf{xor3}_1]\!] \langle 1_2, \langle 1_2, 0_2\rangle\rangle\rangle \rceil_2\\
&= \lceil \langle [\![\mathsf{cond}\,(\mathsf{cond}\,\mathsf{true}\,\mathsf{iden})\,(\mathsf{cond}\,\mathsf{iden}\,\mathsf{false})]\!] \langle 1_2, \langle 1_2, 0_2\rangle\rangle, [\![\mathsf{xor3}_1]\!] \langle 1_2, \langle 1_2, 0_2\rangle\rangle\rangle \rceil_2\\
&= \lceil \langle [\![\mathsf{cond}\,\mathsf{true}\,\mathsf{iden}]\!] \langle 1_2, 0_2\rangle, [\![\mathsf{xor3}_1]\!] \langle 1_2, \langle 1_2, 0_2\rangle\rangle\rangle \rceil_2\\
&= \lceil \langle [\![\mathsf{true}]\!] (0_2), [\![\mathsf{xor3}_1]\!] \langle 1_2, \langle 1_2, 0_2\rangle\rangle\rangle \rceil_2\\
&= \lceil \langle 1_2, [\![\mathsf{xor3}_1]\!] \langle 1_2, \langle 1_2, 0_2\rangle\rangle\rangle \rceil_2\\
&= \lceil \langle 1_2, [\![\mathsf{cond}\,(\mathsf{cond}\,\mathsf{iden}\,(\mathsf{not}\,\mathsf{iden}))\,(\mathsf{cond}\,(\mathsf{not}\,\mathsf{iden})\,\mathsf{iden})]\!] \langle 1_2, \langle 1_2, 0_2\rangle\rangle\rangle \rceil_2\\
&= \lceil \langle 1_2, [\![\mathsf{cond}\,\mathsf{iden}\,(\mathsf{not}\,\mathsf{iden})]\!] \langle 1_2, 0_2\rangle\rangle \rceil_2\\
&= \lceil \langle 1_2, [\![\mathsf{iden}]\!] (0_2)\rangle \rceil_2\\
&= \lceil \langle 1_2, 0_2\rangle \rceil_2\\
&= \lceil 1_2 \rceil_1 \cdot 2^1 + \lceil 0_2 \rceil_1\\
&= 1 \cdot 2 + 0\\
&= 2\\
&= 1 + 0 + 1\\
&= \lceil 1_2 \rceil_1 + \lceil 0_2 \rceil_1 + \lceil 1_2 \rceil_1
\end{aligned}$$

The calculations for the other cases are similar.

Next, we recursively build full adders for any word size.

$$\begin{aligned}&\mathsf{full\text{-}add}_{2n} : 2 \times (2^{2n} \times 2^{2n}) \vdash 2 \times 2^{2n}\\ &\mathsf{full\text{-}add}_{2n} \;:=\; \mathsf{drop}\,(\mathsf{OOH} \vartriangle \mathsf{IOH}) \vartriangle (\mathsf{OH} \vartriangle \mathsf{drop}\,(\mathsf{OIH} \vartriangle \mathsf{IIH})\,;\mathsf{full\text{-}add}_n)\\ &\qquad\qquad ;\; \mathsf{IIH} \vartriangle (\mathsf{IOH} \vartriangle \mathsf{OH}\,;\mathsf{full\text{-}add}_n)\\ &\qquad\qquad ;\; \mathsf{IOH} \vartriangle (\mathsf{IIH} \vartriangle \mathsf{OH})\end{aligned}$$

We generalize the specification of the single bit full adder to the multi-bit full adders.

$$\lceil [\![\mathsf{full\text{-}add}_n]\!] \langle c, \langle a, b\rangle\rangle \rceil_{1,n} \;=\; \lceil a \rceil_n + \lceil b \rceil_n + \lceil c \rceil_1$$

Theorem 3.2. *For all n which is a power of 2, and for all $a : 2^n$, $b : 2^n$, and $c : 2$, we have that* $\lceil [\![\mathsf{full\text{-}add}_n]\!] \langle c, \langle a, b\rangle\rangle \rceil_{1,n} = \lceil a \rceil_n + \lceil b \rceil_n + \lceil c \rceil_1$.

Proof. We prove $\mathsf{full\text{-}add}_n$ meets its specification by induction on n. As mentioned before, the $\mathsf{full\text{-}add}_1$ case is easily checked by verifying all eight possible inputs. Next, we prove that $\mathsf{full\text{-}add}_{2n}$ meets its specification under the assumption that $\mathsf{full\text{-}add}_n$ does. Specifically we need to show that

$$\lceil [\![\mathsf{full\text{-}add}_{2n}]\!] \langle c, \langle \langle a_1, a_2\rangle, \langle b_1, b_2\rangle\rangle\rangle \rceil_{1,2n} = \lceil \langle a_1, a_2\rangle \rceil_{2n} + \lceil \langle b_1, b_2\rangle \rceil_{2n} + \lceil c \rceil_1 \tag{3.1}$$

Let us first consider the right hand side of equation 3.1. By the definition of our value function we have that

$$\begin{aligned}\lceil\langle a_1, a_2\rangle\rceil_{2n} + \lceil\langle b_1, b_2\rangle\rceil_{2n} + \lceil c\rceil_1 &= \lceil a_1\rceil_n \cdot 2^n + \lceil a_2\rceil_n + \lceil b_1\rceil_n \cdot 2^n + \lceil b_2\rceil_n + \lceil c\rceil_1 \\ &= (\lceil a_1\rceil_n + \lceil b_1\rceil_n) \cdot 2^n + \lceil a_2\rceil_n + \lceil b_2\rceil_n + \lceil c\rceil_1\end{aligned}$$

By our inductive hypothesis, we have that

$$\lceil[\![\mathsf{full\text{-}add}_n]\!]\langle c, \langle a_2, b_2\rangle\rangle\rceil_{1,n} = \lceil a_2\rceil_n + \lceil b_2\rceil_n + \lceil c\rceil_1$$

so we know that

$$\lceil\langle a_1, a_2\rangle\rceil_{2n} + \lceil\langle b_1, b_2\rangle\rceil_{2n} + \lceil c\rceil_1 = (\lceil a_1\rceil_n + \lceil b_1\rceil_n) \cdot 2^n + \lceil[\![\mathsf{full\text{-}add}_n]\!]\langle c, \langle a_2, b_2\rangle\rangle\rceil_{1,n}$$

Let us define c_0 and r_0 such that $\langle c_0, r_0\rangle := [\![\mathsf{full\text{-}add}_n]\!]\langle c, \langle a_2, b_2\rangle\rangle$. Thus we have that

$$\begin{aligned}\lceil\langle a_1, a_2\rangle\rceil_{2n} + \lceil\langle b_1, b_2\rangle\rceil_{2n} + \lceil c\rceil_1 &= (\lceil a_1\rceil_n + \lceil b_1\rceil_n) \cdot 2^n + \lceil\langle c_0, r_0\rangle\rceil_{1,n} \\ &= (\lceil a_1\rceil_n + \lceil b_1\rceil_n) \cdot 2^n + \lceil c_0\rceil_1 \cdot 2^n + \lceil r_0\rceil_n \\ &= (\lceil a_1\rceil_n + \lceil b_1\rceil_n + \lceil c_0\rceil_1) \cdot 2^n + \lceil r_0\rceil_n\end{aligned}$$

Again, by our inductive hypothesis, we have that

$$\lceil[\![\mathsf{full\text{-}add}_n]\!]\langle c_0, \langle a_1, b_1\rangle\rangle\rceil_{1,n} = \lceil a_1\rceil_n + \lceil b_1\rceil_n + \lceil c_0\rceil_1$$

therefore we have that

$$\lceil\langle a_1, a_2\rangle\rceil_{2n} + \lceil\langle b_1, b_2\rangle\rceil_{2n} + \lceil c\rceil_1 = \lceil[\![\mathsf{full\text{-}add}_n]\!]\langle c_0, \langle a_1, b_1\rangle\rangle\rceil_{1,n} \cdot 2^n + \lceil r_2\rceil_n$$

Let us define c_1 and r_1 such that $\langle c_1, r_1\rangle := [\![\mathsf{full\text{-}add}_n]\!]\langle\langle a_1, b_1\rangle, c_0\rangle$. Thus we have that

$$\begin{aligned}\lceil\langle a_1, a_2\rangle\rceil_{2n} + \lceil\langle b_1, b_2\rangle\rceil_{2n} + \lceil c\rceil_1 &= \lceil\langle c_1, r_1\rangle\rceil_{1,n} \cdot 2^n + \lceil r_0\rceil_n \\ &= (\lceil c_1\rceil_1 \cdot 2^n + \lceil r_1\rceil_n) \cdot 2^n + \lceil r_0\rceil_n \\ &= \lceil c_1\rceil_1 \cdot 2^{2n} + \lceil r_1\rceil_n \cdot 2^n + \lceil r_0\rceil_n \qquad (3.2)\end{aligned}$$

Now let us consider the left hand side of equation 3.1. By the definition and semantics of $\mathsf{full\text{-}adder}_{2n}$ we have that

$$\begin{aligned}[\![\mathsf{full\text{-}add}_{2n}]\!]\langle c, \langle\langle a_1, a_2\rangle, \langle b_1, b_2\rangle\rangle\rangle &= [\![\mathsf{IOH} \triangle (\mathsf{IIH} \triangle \mathsf{OH})]\!] \\ &\quad \circ [\![\mathsf{IIH} \triangle (\mathsf{IOH} \triangle \mathsf{OH}; \mathsf{full\text{-}add}_n)]\!] \\ &\quad \circ [\![\mathsf{drop}\,(\mathsf{OOH} \triangle \mathsf{IOH}) \triangle (\mathsf{OH} \triangle \mathsf{drop}\,(\mathsf{OIH} \triangle \mathsf{IIH}); \mathsf{full\text{-}add}_n)]\!] \\ &\quad \langle c, \langle\langle a_1, a_2\rangle, \langle b_1, b_2\rangle\rangle\rangle \\ &= [\![\mathsf{IOH} \triangle (\mathsf{IIH} \triangle \mathsf{OH})]\!] \\ &\quad \circ [\![\mathsf{IIH} \triangle (\mathsf{IOH} \triangle \mathsf{OH}; \mathsf{full\text{-}add}_n)]\!] \\ &\quad \langle\langle a_1, b_1\rangle, [\![\mathsf{full\text{-}add}_n]\!]\langle c, \langle a_2, b_2\rangle\rangle\rangle \\ &= [\![\mathsf{IOH} \triangle (\mathsf{IIH} \triangle \mathsf{OH})]\!] \\ &\quad \circ [\![\mathsf{IIH} \triangle (\mathsf{IOH} \triangle \mathsf{OH}; \mathsf{full\text{-}add}_n)]\!] \\ &\quad \langle\langle a_1, b_1\rangle, \langle c_0, r_0\rangle\rangle \\ &= [\![\mathsf{IOH} \triangle (\mathsf{IIH} \triangle \mathsf{OH})]\!] \\ &\quad \langle r_0, [\![\mathsf{full\text{-}adder}_n]\!]\langle c_0, \langle a_1, b_1\rangle\rangle\rangle \\ &= [\![\mathsf{IOH} \triangle (\mathsf{IIH} \triangle \mathsf{OH})]\!] \\ &\quad \langle r_0, \langle c_1, r_1\rangle\rangle \\ &= \langle c_1, \langle r_1, r_0\rangle\rangle\end{aligned}$$

Therefore we have that

$$\begin{aligned}\lceil[\![\mathsf{full\text{-}add}_{2n}]\!]\langle c, \langle\langle a_1, a_2\rangle, \langle b_1, b_2\rangle\rangle\rangle\rceil_{1,2n} &= \lceil\langle c_1, \langle r_1, r_0\rangle\rangle\rceil_{1,2n} \\ &= \lceil c_1\rceil_1 \cdot 2^{2n} + \lceil\langle r_1, r_0\rangle\rceil_{2n} \\ &= \lceil c_1\rceil_1 \cdot 2^{2n} + \lceil r_1\rceil_n \cdot 2^n + \lceil r_0\rceil_n \qquad (3.3)\end{aligned}$$

Together equations 3.2 and 3.3 show that the right hand side and left hand side of equation 3.1 are equal, as required. □

Computer verified versions of theses proofs can be found in the Coq library (see Section 8.3.2).
With a full adder we can recursively build full multipliers.

$$\begin{aligned}
&\mathsf{full\text{-}multiply}_1 : (2 \times 2) \times (2 \times 2) \vdash 2^2 \\
&\mathsf{full\text{-}multiply}_1 \;:=\; \mathsf{take}\,(\mathsf{cond\ iden\ false}) \vartriangle \mathsf{IH}; \mathsf{full\text{-}add}_1
\end{aligned}$$

$$\begin{aligned}
&\mathsf{full\text{-}multiply}_{2n} : (2^{2n} \times 2^{2n}) \times (2^{2n} \times 2^{2n}) \vdash 2^{4n} \\
&\mathsf{full\text{-}multiply}_{2n} \;:=\; \mathsf{take}\,(\mathsf{OOH} \vartriangle (\mathsf{IOH} \vartriangle \mathsf{OIH})) \\
&\qquad\qquad \vartriangle\; ((\mathsf{take}\,(\mathsf{OOH} \vartriangle \mathsf{IIH}) \vartriangle \mathsf{drop}\,(\mathsf{OOH} \vartriangle \mathsf{IOH}); \mathsf{full\text{-}multiply}_n) \\
&\qquad\qquad \vartriangle\; (\mathsf{take}\,(\mathsf{OIH} \vartriangle \mathsf{IIH}) \vartriangle \mathsf{drop}\,(\mathsf{OIH} \vartriangle \mathsf{IOH}); \mathsf{full\text{-}multiply}_n)) \\
&\qquad\qquad ;\; \mathsf{take}\,(\mathsf{OH} \vartriangle \mathsf{IOH}) \\
&\qquad\qquad \vartriangle\; (\mathsf{drop}\,(\mathsf{OOH} \vartriangle \mathsf{IIH}) \vartriangle (\mathsf{OIH} \vartriangle \mathsf{drop}\,(\mathsf{OIH} \vartriangle \mathsf{IOH}); \mathsf{full\text{-}multiply}_n)) \\
&\qquad\qquad ;\; (\mathsf{OH} \vartriangle \mathsf{drop}\,(\mathsf{IOH} \vartriangle \mathsf{OOH}); \mathsf{full\text{-}multiply}_n) \vartriangle \mathsf{drop}\,(\mathsf{IIH} \vartriangle \mathsf{OIH})
\end{aligned}$$

We can prove that the full multipliers meet the following specifications.

$$\lceil [\![\mathsf{full\text{-}multiply}_n]\!] \langle \langle a, b \rangle, \langle c, d \rangle \rangle \rceil_{2n} \;=\; \lceil a \rceil_n \cdot \lceil b \rceil_n + \lceil c \rceil_n + \lceil d \rceil_n$$

TODO: Notes on trade-offs between efficiency and simplicity.

3.3.5 Bitwise Operations

3.3.6 SHA-256

The official standard for the SHA-2 family, which includes SHA-256, can be found in the FIPS PUB 180-4: Secure Hash Standard (SHS) [?]. We define the SHA-256 function, $\mathrm{SHA256}_2 : 2^* \to 2^{256}$, as a function from bit strings to a 256-bit word. Technically, SHA-256 is restricted to inputs $l : 2^*$ where $|l| < 2^{64}$.

The SHA-256 hash function is composed from two components, a padding function $\mathrm{SHA256}_{\mathrm{Pad}} : 2^* \to (2^{512})^+$, which appends padding and length data to produce a (non-empty) sequence of blocks of 512 bits, and the Merkle–Damgård construction $\mathrm{SHA256}_{\mathrm{MD}} : 2^{256} \times (2^{512})^* \to 2^{256}$.

$$\mathrm{SHA256}_2(l) = \mathrm{SHA256}_{\mathrm{MD}} \langle \mathrm{SHA256}_{\mathrm{IV}}, \eta^S(\mathrm{SHA256}_{\mathrm{Pad}}(l)) \rangle$$

where $\mathrm{SHA256}_{\mathrm{IV}} : 2^{256}$ is the SHA-256 initial value and $\eta^S_{A^+} : A^+ \to A^*$ formally converts a non-empty list to a regular list.

The $\mathrm{SHA256}_{\mathrm{MD}}$ function is a left fold using the SHA-256 block compression function $\mathrm{SHA256}_{\mathrm{Block}} : 2^{256} \times 2^{512} \to 2^{256}$.

$$\begin{aligned}
\mathrm{SHA256}_{\mathrm{MD}} \langle h, \epsilon \rangle &:= h \\
\mathrm{SHA256}_{\mathrm{MD}} \langle h, b \blacktriangleleft l \rangle &:= \mathrm{SHA256}_{\mathrm{MD}} \langle \mathrm{SHA256}_{\mathrm{Block}} \langle h, b \rangle, l \rangle
\end{aligned}$$

The block compression function $\mathrm{SHA256}_{\mathrm{Block}} : 2^{256} \times 2^{512} \to 2^{256}$ is a function whose type fits in Simplicity's framework. We can create a core Simplicity term $\mathsf{sha256\text{-}block} : 2^{256} \times 2^{512} \vdash 2^{256}$ that implements this function

$$[\![\mathsf{sha256\text{-}block}]\!] = \mathrm{SHA256}_{\mathrm{Block}}$$

We can also define SHA-256's initial value $\mathsf{sha256\text{-}iv} : \mathbb{1} \vdash 2^{256}$.

$$[\![\mathsf{sha256\text{-}iv}]\!] \langle\rangle = \mathrm{SHA256}_{\mathrm{IV}}$$

Beyond defining the block compression function in the Simplicity language, we will also be using the SHA-256 hash function elsewhere in this specification. In practice, SHA-256 is applied to byte strings rather than bit strings. To this end, we define the variant $\mathrm{SHA256}_{2^8} : (2^8)^* \to 2^{256}$.

$$\mathrm{SHA256}_{2^8} := \mathrm{SHA256}_2 \circ \mu^* \circ (\iota^{2^8}_{2^*})^*$$

where $\iota^{2^8}_{2^*} : 2^8 \to 2^*$ formally converts a byte to a bit string (in big endian format).

$$\iota^{2^8}_{2^*}\langle\langle\langle b_0, b_1\rangle, \langle b_2, b_3\rangle\rangle, \langle\langle b_4, b_5\rangle, \langle b_6, b_7\rangle\rangle\rangle := b_0 \blacktriangleleft b_1 \blacktriangleleft b_2 \blacktriangleleft b_3 \blacktriangleleft b_4 \blacktriangleleft b_5 \blacktriangleleft b_6 \blacktriangleleft b_7 \blacktriangleleft \epsilon$$

Since the $\mathrm{SHA256}_{2^8}$ variant is so commonly used, we will write it unadorned as simply SHA256.

3.3.6.1 Tagged Hashes

BIP-340, BIP-341 used a "tagged-hash" format to help avoid message collisions between different message variants. We adopt the same format here to avoid message collisions within a Simplicity application and we hope to avoid message collision across various applications.

The tagged hash format begins every message with one block prefix which indicate the variant, or type of digest data being hashed. This first block consists of the SHA-256 digest of a string repeated twice. Given a byte string $t : (2^8)^*$, we can defined the tagged hash of bit strings, $\mathrm{SHA256}^t_2 : 2^* \to 2^{256}$, and byte strings, $\mathrm{SHA256}^t_{2^8} : (2^8)^* \to 2^{256}$.

$$\begin{array}{rcl} \mathrm{SHA256}^t_2(x) & := & \mathrm{SHA256}_2(\mu^*((\iota^{2^8}_{2^*})^*(p{\cdot}p)){\cdot}x) \\ \mathrm{SHA256}^t_{2^8}(x) & := & \mathrm{SHA256}_{2^8}(p{\cdot}p{\cdot}x) \\ & \text{where} & p := \mathrm{BE}_{256}(\mathrm{SHA256}_{2^8}(t)) \end{array}$$

We can compute the SHA-256 midstate after compressing this first block containing the prefix tag.

$$\mathrm{SHA256}^t_{\mathrm{IV}} := \mathrm{SHA256}_{\mathrm{Block}}\langle \mathrm{SHA256}_{\mathrm{IV}}, \Delta(\mathrm{SHA256}_{2^8}(t))\rangle$$

3.3.7 Elliptic Curve Operations on secp256k1

The Standards for Efficient Cryptography (SEC) documents have recommend modular elliptic curve parameters including the secp256k1 curve [?], which is used by Bitcoin's EC-DSA signature scheme and the proposed EC-Schnorr scheme.

Most points on an elliptic curve, such as secp256k1, consist of a pair of coordinates from a specified finite field. In the case of secp256k1, the finite field is the prime field $\mathbb{F}_p$ where $p := 2^{256} - 4294968273$. The elliptic curve for secp256k1 consists of the points $\langle x, y\rangle$ satisfying the equation $y^2 \equiv x^3 + 7 \pmod p$, plus an additional "point at infinity", which we will write as $\mathcal{O}$. It turns out that elliptic curves can be given a group structure via "geometry", where any three points on the curve that are co-linear sum to 0 under this group structure, and where $\mathcal{O}$ is the group's identity element. We have to be careful to count lines "tangent" to the curve as passing through the same point twice, and count vertical lines as passing through $\mathcal{O}$. This group structure is Abelian, and therefore can also be viewed as a $\mathbb{Z}_n$-module[3.1] where $n := 2^{256} - 432420386565659656852420866394968145599$ is the order of this elliptic curve. This $\mathbb{Z}_n$-module structure allows us to talk about "adding" two points of the elliptic curve, and scaling a point by a factor from $\mathbb{Z}_n$.

Because the order of the elliptic curve, n, is a prime number, every non-$\mathcal{O}$ element generates the entire curve (through scalar multiplication). The specification for secp256k1 comes with a reference generator, $\mathcal{G}$, which is defined as the following point.

$$\begin{array}{rcl} \mathcal{G} & := & \langle 55066263022277343669578718895168534326250603453777594175500187360389116729240 \\ & & , 32670510020758816978083085130507043184471273380659243275938904335757337482424\rangle \end{array}$$

3.3.7.1 libsecp256k1

The libsecp256k1 library [?] is a C implementation of optimized functions on this elliptic curve. This library has two variants for the representation of elliptic curve point values. The affine coordinate representation consists of a pair of field elements, and a flag to indicate the value $\mathcal{O}$ (in which case the coordinate values are ignored). The Jacobian coordinate representation consists of a triple of field elements, and a flag to indicate the value $\mathcal{O}$ (in which case the coordinates are ignored).

3.1. Mathematically, we actually have a 1-dimensional vector space; however determining the linear dependence between two vectors is presumed to be infeasible. Indeed, the security properties of the elliptic curve depends on this being infeasible. For this reason, it is more useful to think of this structure as a module rather than as a vector space.

A point in Jacobian coordinates, $\langle x, y, z\rangle$ is defined to be on the elliptic curve when

$$y^2 \equiv x^3 + 7z^6 \pmod{p}$$

and two points in Jacobian coordinates are equivalent, $\langle x_0, y_0, z_0\rangle \asymp \langle x_1, y_1, z_1\rangle$, when

$$x_0\, z_1^2 \equiv x_1\, z_0^2 \pmod{p} \text{ and } y_0\, z_1^3 \equiv y_1\, z_0^3 \pmod{p}.$$

A point in Jacobian coordinates, $\langle x,\ y,\ z\rangle$ represents the curve point $\langle x\ z^{-2},\ y\ z^{-3}\rangle$ in affine coordinates when $z \neq 0$. In particular, the point $\langle x,\ y,\ 1\rangle$ in Jacobian coordinates represents the point $\langle x,\ y\rangle$ in affine coordinates. The same point has multiple representations in Jacobian coordinates, however, even the affine coordinate representation is redundant because the underlying field representation is itself redundant.

Normally the point at infinity would be represented by $\langle a^2, a^3, 0\rangle$ in Jacobian coordinates for any $a \in \mathbb{F}_p$; however this is not done in libsecp256k1. Instead a flag is used to represent the point at infinity (and the coordinates are ignored when this flag is set). Testing if a field element is equivalent to 0 is a non-trivial operation, so using a flag like this may be sensible.

The various group and $\mathbb{Z}_n$-module operations are implemented efficiently, again subject to various specific preconditions on their inputs. In particular, the operation for forming linear combinations of the form

$$\sum_{i=0}^{k} n_{\mathcal{A}_i}\, \mathcal{A}_i + n_{\mathcal{G}}\, \mathcal{G}$$

is supported using an algorithm known as Shamir's trick, where $n_{\mathcal{A}_i} : \mathbb{Z}_n$, $n_{\mathcal{G}} : \mathbb{Z}_n$, and $\mathcal{A}_i$ are points on the elliptic curve.

3.3.7.2 libsecp256k1 in Simplicity

The primary application for Simplicity is to implement computation for public validation. In implementing cryptographic operations in Simplicity, we have no need to worry about constant-time implementations, nor securing private key material because Simplicity's application only processes public data.

When it comes to implementing elliptic curve operations in Simplicity, we do face one problem. In order for elliptic curve operations to be fast, we need a representation of field elements and curve points that have redundant representations, but then the choice of which specific representative returned by Simplicity expressions becomes consensus critical.

We see three possible ways of addressing this problem:

1. We can define minimal Simplicity types that can represent field elements and elliptic curve points and return values in normal form after every elliptic curve operation.
2. We can define Simplicity types that can represent field elements and elliptic curve points with redundant representations and specify precisely which representative is the result of each elliptic curve operation.
3. We can extend Simplicity with abstract data types for field elements and elliptic curve points and enforce data abstraction in the elliptic curve operations.

Option 1 would make it easy for developers to implement elliptic curve jets (see Section ⟨reference | TODO⟩) that short-cut the interpretation of Simplicity's elliptic curve operations by running native code instead. The native code for these jets can be implemented by any reasonable algorithm, and the results normalized. The problem with this option is that computing the normal form of elliptic curve points is an expensive operation. In particular, normalizing a point in Jacobian coordinates requires computing a modular inverse, which is a very expensive operation compared to the other field operations.

Option 2 means we must ensure that the native code for jets returns exactly the same representative that the Simplicity expression it replaces would have produced. Even libsecp256k1 does not guarantee that different versions of the library will return the same representatives for its basic elliptic curve operations, so Simplicity jets would not be able to keep up with libsecp256k1 updates.

Option 3 would let us change the underlying representations of elliptic curve values, allowing us to use any version of any secp256k1 library. However, it would extend the scope of Simplicity beyond what we are willing to do.

We have chosen to go with option 2. We have reimplemented the exact same algorithms for field and elliptic curve operations that the most recent release of libsecp256k1 uses as of the time of this writing, including computing of linear combinations of the form

$$n_{\mathcal{A}}\,\mathcal{A} + n_{\mathcal{G}}\,\mathcal{G}$$

which is used for Schnorr signature validation. Our jets will be tied to this specific version of libsecp256k1, and the commitment Merkle root (see Section 3.7) captures the formal specification of the functional behaviour of our jets. The libsecp256k1 is already reasonably mature, so we are not expecting to lose out too much by missing future advances. When there are major improvements, new versions of Simplicity jets could be substituted in by using a versioning mechanism for Simplicity.

In Simplicity, we represent a field element by the type

$$\mathrm{FE} := 2^{256}$$

and a value $a : \mathrm{FE}$ represents the field element

$$\lceil a \rceil_{\mathrm{FE}} := \lceil a \rceil_{256} \pmod p.$$

We represent non-$\mathcal{O}$ points on the secp256k1 elliptic curve in affine coordinates by the type

$$\mathrm{GE} := \mathrm{FE} \times \mathrm{FE}$$

and a value $\langle x, y\rangle : \mathrm{GE}$ represents the point

$$\lceil \langle x, y\rangle \rceil_{\mathrm{GE}} := \langle \lceil x \rceil_{\mathrm{FE}}, \lceil y \rceil_{\mathrm{FE}}\rangle.$$

We also represent points on the secp256k1 elliptic curve in Jacobian coordinates by the type

$$\mathrm{GEJ} := \mathrm{GE} \times \mathrm{FE}$$

and a value $\langle\langle x, y\rangle, z\rangle$ represents the point

$$\begin{array}{rcll} \lceil\langle\langle x, y\rangle, z\rangle\rceil_{\mathrm{GEJ}} & := & \langle \lceil x \rceil_{\mathrm{FE}} \cdot \lceil z \rceil_{\mathrm{FE}}^{-2} \pmod p, \lceil y \rceil_{\mathrm{FE}} \cdot \lceil z \rceil_{\mathrm{FE}}^{-3} \pmod p\rangle & \text{when } \lceil z \rceil_{\mathrm{FE}} \not\equiv 0 \pmod p \\ \lceil\langle\langle x, y\rangle, z\rangle\rceil_{\mathrm{GEJ}} & := & \mathcal{O} & \text{when } \lceil z \rceil_{\mathrm{FE}} \equiv 0 \pmod p \text{ and } \lceil y \rceil_{\mathrm{FE}}^2 \equiv \lceil x \rceil_{\mathrm{FE}}^3 \pmod p. \end{array}$$

The translation between libsecp256k1's affine coordinate representation of elliptic curve points and Simplicity's GE is straightforward except that Simplicity's GE type has no flag and cannot represent the $\mathcal{O}$ point. The translation between libsecp256k1's Jacobian coordinate representation of elliptic curve points and Simplicity's GEJ type is mostly straight forward, however the GEJ type represents the $\mathcal{O}$ point using a z-coordinate representing 0, while libsecp256k1 uses a flag to represent the $\mathcal{O}$ point.

The Simplicity implementation of libsecp256k1 is designed so that libsecp256k1 can be used as jets for these Simplicity expressions. As such, the Simplicity expressions are designed to mimic the exact behaviour of a specific version of libsecp256k1's elliptic curve functions. For inputs of particular representations of point in Jacobian coordinates, the Simplicity expression returns the exact same representative for its result as libsecp256k1. If an off-curve point is passed to a libsecp256k1 function, the Simplicity code again computes the same result that the libsecp256k1 function does.

The only subtle point with using libsecp256k1 for jets lies in the different representation of $\mathcal{O}$. The inputs and outputs of operations need to be suitable translated between the two representations. However, this can be done as part of the marshalling code for the jets, and the Simplicity expressions are written with this in mind.

3.3.7.3 Schnorr Signature Validation

With elliptic curve operations defined, we are able to implement Schnorr signature validation in accordance with the BIP-0340 specification [?]. We define Simplicity types for the formats of x-only public keys, PubKey; messages, Msg; and Schnorr signatures, Sig, below.

$$\begin{array}{rcl} \mathrm{PubKey} & := & 2^{256} \\ \mathrm{Msg} & := & 2^{256} \\ \mathrm{Sig} & := & 2^{512} \end{array}$$

The PubKey type is an x-coordinate of a point whoes y-coordinate's least significant bit is even. A Msg value m represents the byte-string $\mathrm{BE}_{256}(m)$ for a Schnorr signature's message, and a Sig value a represents the byte-string $\mathrm{BE}_{512}(a)$ for a Schnorr signature.

We have implemented a core Simplicity expression to check a Schnorr signature for a public key on a given message:

$$\mathsf{bip\text{-}0340\text{-}check} : (\mathrm{PubKey} \times \mathrm{Msg}) \times \mathrm{Sig} \vdash 2$$

The semantics are such that $[\![\mathsf{bip\text{-}0340\text{-}check}]\!]\langle\langle p, m\rangle, s\rangle = \mathtt{1}_2$ only when the values that the inputs represents satisfy the verification conditions of the BIP-0340 specification.

3.4 Completeness Theorem

General purpose programming languages are famously incomplete because there are functions that are uncomputable, the halting problem being the most famous of these. Core Simplicity is even more limited that these general purpose programming languages because its denotational semantics are limited to functions from finite types to finite types.

However, we can ask the question, is every function from a finite type to a finite type expressible in core Simplicity? This question is answered by the following completeness theorem.

Theorem 3.3. *Core Simplicity Completeness Theorem. For any Simplicity types A and B and any function $f : A \to B$, there exists some core Simplicity term t such that for all $a : A$,*

$$[\![t]\!](a) = f(a)$$

This result is possible because these functions are all finitary and can be, in principle, expressed as a large lookup table. It is possible to encode these lookup tables with Simplicity expressions. The formal proof of this theorem can be found in the Coq library (see Section 8.2.1).

It is worth emphasizing that this result is a purely theoretical result that shows that core Simplicity is fully expressive for its domain; it is completely impractical to generate Simplicity expressions this way as many expressions would be astronomical in size. Thus we can view Simplicity programming as an exercise in compression: how can we take advantage of the structure within computations to express our functions succinctly.

One practical variant of the core Simplicity completeness theorem is that for any value of a Simplicity type $b : B$, the constant function $\lambda\_.\, b : A \to B$ can be realized by a Simplicity expression. We call the function that constructs this term, $\mathsf{scribe}_{A,B}(b) : A \vdash B$.

$$\begin{aligned}
\mathsf{scribe}_{A,\mathbb{1}}\langle\rangle &:= \mathsf{unit} \\
\mathsf{scribe}_{A,B+C}(\sigma^{\mathbf{L}}_{B,C}(b)) &:= \mathsf{injl}\,(\mathsf{scribe}_{A,B}(b)) \\
\mathsf{scribe}_{A,B+C}(\sigma^{\mathbf{R}}_{B,C}(c)) &:= \mathsf{injr}\,(\mathsf{scribe}_{A,C}(c)) \\
\mathsf{scribe}_{A,B\times C}\langle b, c\rangle &:= \mathsf{scribe}_{A,B}(b) \vartriangle \mathsf{scribe}_{A,C}(c)
\end{aligned}$$

Theorem 3.4. *For all Simplicity types A and B, and for all values $a : A$ and $b : B$, $[\![\mathsf{scribe}_{A,B}(b)]\!](a) = b$.*

3.5 Operational Semantics

The denotational semantics of Simplicity determine the functional behaviour of expressions. However, they are not suitable for determining the computation resources needed to evaluate expressions. For this reason we define an operational semantics for Simplicity via an abstract machine we call the *Bit Machine*.

3.5.1 Representing Values as Cell Arrays

Values in the Bit Machine are represented by arrays of cells where each cell contains one of three values: a `0` value, a `1` value, or a `?` value which we call an undefined value. We write an array of cells by enclosing a sequence of cells with square brackets (e.g. `[1?0]`). We denote the length of an array using $|\cdot|$. For example, $|\texttt{[1?0]}| = 3$. The concatenation of two arrays, a and b is denoted by $a \cdot b$, and replication of an array n times is denoted by exponentiation, a^n. Sometimes we will omit the dot when performing concatenation.

For any given type, we define the number of cells needed to hold values of that type using the following bitSize function.

$$\begin{aligned} \mathrm{bitSize}(\mathbb{1}) &:= 0 \\ \mathrm{bitSize}(A + B) &:= 1 + \max\left(\mathrm{bitSize}(A), \mathrm{bitSize}(B)\right) \\ \mathrm{bitSize}(A \times B) &:= \mathrm{bitSize}(A) + \mathrm{bitSize}(B) \end{aligned}$$

We define a representation of values of Simplicity types as arrays of cells as follows.

$$\begin{aligned} \ulcorner \langle\rangle \urcorner_{\mathbb{1}} &:= \texttt{[]} \\ \ulcorner \sigma^{\mathbf{L}}_{A,B}(a) \urcorner_{A+B} &:= \texttt{[0]} \cdot \texttt{[?]}^{\mathrm{padL}(A,B)} \cdot \ulcorner a \urcorner_A \\ \ulcorner \sigma^{\mathbf{R}}_{A,B}(b) \urcorner_{A+B} &:= \texttt{[1]} \cdot \texttt{[?]}^{\mathrm{padR}(A,B)} \cdot \ulcorner b \urcorner_B \\ \ulcorner \langle a, b \rangle \urcorner_{A \times B} &:= \ulcorner a \urcorner_A \cdot \ulcorner b \urcorner_B \end{aligned}$$

The representation of values of a sum type are padded with undefined cells so that the representation has the proper length.

$$\begin{aligned} \mathrm{padL}(A, B) &:= \max\left(\mathrm{bitSize}(A), \mathrm{bitSize}(B)\right) - \mathrm{bitSize}(A) \\ \mathrm{padR}(A, B) &:= \max\left(\mathrm{bitSize}(A), \mathrm{bitSize}(B)\right) - \mathrm{bitSize}(B) \end{aligned}$$

Theorem 3.5. *Given any value of some Simplicity type, $a : A$, we have $|\ulcorner a \urcorner_A| = \mathrm{bitSize}(A)$.*

3.5.2 Bit Machine

A frame is a, possibly empty, cell array with a cursor referencing a cell in the array, which we denote using an underscore.

$$\texttt{[01}\underline{\texttt{1}}\texttt{?10]}$$

The cursor may also reference the end of the array, which we denote by marking the end of the array with an underscore.

$$\texttt{[01?10}\underline{\texttt{]}}$$

Frames can be concatenated with cell arrays either on the left or on the right without moving the cursor. Note that when concatenating a non-empty cell array onto the right hand side of a frame whose cursor is at the end of the frame, the cursor ends up pointing to the first cell of the added cell array.

$$\texttt{[01?10}\underline{\texttt{]}}\texttt{[111??]} = \texttt{[01?10}\underline{\texttt{1}}\texttt{11??]}$$

We will sometimes denote the empty frame, `[]`, with a small cursor, $\wedge$.

The state of the Bit Machine consists of two non-empty stacks of frames: a read-frame stack and a write-frame stack. The top elements of the two stacks are called the *active read frame* and the *active write frame* respectively. The other frames are called inactive read-frames and inactive write-frames.

read frame stack	write frame stack
`[1001`$\underline{\texttt{1}}$`1??110101000]`	`[11??1101`$\underline{\texttt{]}}$
`[`$\underline{\texttt{0}}$`000]`	`[111`$\underline{\texttt{?}}$`?]`
`[`$\underline{\texttt{]}}$	
`[`$\underline{\texttt{1}}$`0]`	

Figure 3.1. Example state of the Bit Machine.

Notationally we will write a stack of read frames as $r_n \rhd \ldots \rhd r_1 \rhd r_0$, with r_0 as the active read frame. We will write a stack of write frames in the opposite order, as $w_0 \lhd w_1 \lhd \ldots \lhd w_m$ with w_0 as the active write frame. We write a state of the Bit Machine as $[\Theta \rhd r_0 | w_0 \lhd \Xi]$ where Θ is the (possibly empty) inactive read frame stack, Ξ is the (possibly empty) inactive write frame stack, r_0 is the active read frame, and w_0 is the active write frame.[3.2] There is one additional state of the Bit Machine called the *halted* state, which we denote by $\boxtimes$.

The Bit Machine has nine basic instructions that, when executed, transform the Bit Machine's state. We denote these basic instructions as $i : S_0 \rightsquigarrow S_1$, where i is the instructions's name, S_0 is a state of the Bit Machine before executing the instruction, and S_1 is the state of the machine after the successful execution of the instructions.

3.5.2.1 Frame Instructions

Our first three basic instructions, create, move, and delete active frames.

$$\begin{aligned}
\text{newFrame}(n) &: [\Theta \rhd r_0 | w_0 \lhd \Xi] \rightsquigarrow [\Theta \rhd r_0 | \wedge[?]^n \lhd w_0 \lhd \Xi] \\
\text{moveFrame} &: [\Theta \rhd r_0 | [c_1 \mathbin{|}\cdots\mathbin{|} c_n] \lhd w_0 \lhd \Xi] \rightsquigarrow [\Theta \rhd r_0 \rhd [c_1 \cdots c_n] | w_0 \lhd \Xi] \\
\text{dropFrame} &: [\Theta \rhd r_1 \rhd r_0 | \Xi] \rightsquigarrow [\Theta \rhd r_1 | \Xi]
\end{aligned}$$

Executing a newFrame(n) instruction pushes a new frame of length n onto the write frame stack. This new frame has its cursor at the beginning of the frame and the entire frame is filled with undefined values. It is legal for the new frame to have length 0.

Executing the moveFrame instruction moves the top frame of the write frame stack to the read frame stack. This instruction is only legal to execute when the cursor of the active write frame is at the end of the frame. The cursor is reset to the beginning of the frame when it is placed onto the read frame stack.

Executing the dropFrame instruction removes the top frame of the read frame stack.

3.5.2.2 Active Write Frame Instructions

Our next three instructions operate on the active write frame.

$$\begin{aligned}
\text{write}(0) &: [\Theta \rhd r_0 | w_0 \cdot [?] [?]^m \lhd \Xi] \rightsquigarrow [\Theta \rhd r_0 | w_0 \cdot [0] [?]^m \lhd \Xi] \\
\text{write}(1) &: [\Theta \rhd r_0 | w_0 \cdot [?] [?]^m \lhd \Xi] \rightsquigarrow [\Theta \rhd r_0 | w_0 \cdot [1] [?]^m \lhd \Xi] \\
\text{skip}(n) &: [\Theta \rhd r_0 | w_0 \wedge [?]^{n+m} \lhd \Xi] \rightsquigarrow [\Theta \rhd r_0 | w_0 \cdot [?]^n \wedge [?]^m \lhd \Xi] \\
\text{copy}(n) &: [\Theta \rhd r_0 \cdot [c_1 \cdots c_n] \cdot r_0' | w_0 \wedge [?]^{n+m} \lhd \Xi] \rightsquigarrow [\Theta \rhd r_0 \cdot [c_1 \cdots c_n] \cdot r_0' | w_0 \cdot [c_1 \mathbin{|}\cdots\mathbin{|} c_n] [?]^m \lhd \Xi]
\end{aligned}$$

Executing a write(b) instruction writes a 0 or 1 to the active write frame and advances its cursor. Writing an undefined value using this instruction is not allowed. The cursor cannot be at the end of the frame.

Executing a skip(n) instruction advances the active write frame's cursor without writing any data. There must be sufficient number of cells after the cursor. The trivial instruction skip(0) is legal and executing it is effectively a no-op.

Executing a copy(n) instruction copies the values of the n cells after the active read frame's cursor into the active write frame, advancing the write frame's cursor. The must be a sufficient number of cells after both the active read frame and active write frame's cursors. Note that undefined cell values are legal to copy. The trivial instruction copy(0) is legal and executing it is effectively a no-op.

3.5.2.3 Active Read Frame Instructions

The next two instructions are used to manipulate the active read frame's cursor.

$$\text{fwd}(n) : [\Theta \rhd r_0 \cdot [c_1 \cdots c_n] \cdot r_0' | w_0 \lhd \Xi] \rightsquigarrow [\Theta \rhd r_0 \cdot [c_1 \mathbin{|}\cdots\mathbin{|} c_n] \cdot r_0' | w_0 \lhd \Xi]$$

$$\text{bwd}(n) : [\Theta \rhd r_0 \cdot [c_1 \mathbin{|}\cdots\mathbin{|} c_n] \cdot r_0' | w_0 \lhd \Xi] \rightsquigarrow [\Theta \rhd r_0 \cdot [c_1 \cdots c_n] \cdot r_0' | w_0 \lhd \Xi]$$

3.2. The notation for the Bit Machine's state is intended to mimic the gap buffer used in our C implementation of the Bit Machine (see TODO: C implementation).

Executing a fwd(n) instructions moves the cursor on the active read frame forward, and executing a bwd(n) instruction moves the cursor backwards. In both cases there must be sufficient number of cells before or after the cursor. The trivial instructions fwd(0) and bwd(0) are legal and executing them are effectively no-ops.

3.5.2.4 Abort Instruction

The final instruction of for the Bit Machine moves from any non-halted state into the halted state.

$$\text{abort} : [\Theta \rhd r_0 | w_0 \lhd \Xi] \rightsquigarrow \boxtimes$$

This is the only way to enter the halted state, and once in the halted state no further instructions can be executed.

3.5.2.5 Bit Machine Programs

The basic instructions of the Bit Machine are combined to produce programs that take the Bit Machine through a sequence of states. We write $S_0 \twoheadrightarrow k \twoheadrightarrow S_1$ for a program, k, that, when executed, successfully transforms an initial state S_0 to the final state S_1.

$$\frac{}{S \twoheadrightarrow \text{nop} \twoheadrightarrow S}$$

We write nop for the trivial program with no instructions. The initial and final states are identical in this case.

$$\frac{i : S_0 \rightsquigarrow S_1}{S_0 \twoheadrightarrow i \twoheadrightarrow S_1}$$

For every basic instruction there is a single instruction program whose initial and final states match those of the basic instruction.

$$\frac{S_0 \twoheadrightarrow k_0 \twoheadrightarrow S_1 \quad S_1 \twoheadrightarrow k_1 \twoheadrightarrow S_2}{S_0 \twoheadrightarrow k_0; k_1 \twoheadrightarrow S_2}$$

We write $k_0; k_1$ for a sequence of two programs, k_0 and k_1. The Bit Machine executes the two programs in turn, concatenating the sequence of states of the two programs.

$$\frac{[\Theta \rhd r_0 \cdot \texttt{[}\underline{\texttt{0}}\texttt{]} \cdot r_0' | w_0 \lhd \Xi] \twoheadrightarrow k_0 \twoheadrightarrow S}{[\Theta \rhd r_0 \cdot \texttt{[}\underline{\texttt{0}}\texttt{]} \cdot r_0' | w_0 \lhd \Xi] \twoheadrightarrow k_0 || k_1 \twoheadrightarrow S}$$

$$\frac{[\Theta \rhd r_0 \cdot \texttt{[}\underline{\texttt{1}}\texttt{]} \cdot r_0' | w_0 \lhd \Xi] \twoheadrightarrow k_1 \twoheadrightarrow S}{[\Theta \rhd r_0 \cdot \texttt{[}\underline{\texttt{1}}\texttt{]} \cdot r_0' | w_0 \lhd \Xi] \twoheadrightarrow k_0 || k_1 \twoheadrightarrow S}$$

We define $k_0 || k_1$ as a deterministic choice between two programs, k_0 and k_1. When executing a deterministic choice, the value under the active read frame's cursor decides which one of the two programs are executed. When encountering a deterministic choice, the active read frame's cursor must not be at the end of its array and the cell under the cursor must not be an undefined value.

$$\frac{}{\boxtimes \twoheadrightarrow k \twoheadrightarrow \boxtimes}$$

Lastly, we stipulate that every program when executed from the halted state ignores all instructions and perform a no-op instead.

Take care to note this difference between instructions and programs containing one instruction. A single instruction cannot be executed starting from the halted state, while a program that consists of a single instruction can be run starting from the halted state (however, it does nothing from this state).

$$n \star k := \text{fwd}(n); k; \text{bwd}(n)$$

The $n \star k$ notation (called "bump") is for a program that temporarily advances the active read frame's cursor when executing k.

Theorem 3.6.

$$\frac{[\Theta \rhd r_0 \cdot \texttt{[}c_1 \cdots c_n\texttt{]} \cdot r_0' | w_0 \lhd \Xi] \twoheadrightarrow k \twoheadrightarrow [\Theta \rhd r_0 \cdot \texttt{[}c_1 \cdots c_n\texttt{]} \cdot r_0' | w_0' \lhd \Xi']}{[\Theta \rhd r_0 \cdot \texttt{[}c_1 \cdots c_n\texttt{]} \cdot r_0' | w_0 \lhd \Xi] \twoheadrightarrow n \star k \twoheadrightarrow [\Theta \rhd r_0 \cdot \texttt{[}c_1 \cdots c_n\texttt{]} \cdot r_0' | w_0' \lhd \Xi']}$$

3.5.2.6 Crashing the Bit Machine

Bit Machine programs are deterministic. Given a program k and an initial state S_0 there exists at most one state S_1 such that $S_0 \rightarrowtail k \twoheadrightarrow S_1$. However it is possible that there is no state S_1 such that $S_0 \rightarrowtail k \twoheadrightarrow S_1$ given an initial state for a program. This happens when the Bit Machine is trying to execute a single instruction program, i, from a non-halted state where that instruction cannot legally execute from. This can also happen when a deterministic choice operation is encountered starting from a state where the active read frame's cursor is at the end of the frame, or is referencing and undefined value.

When a program cannot execute to completion from a given initial state, we say that the Bit Machine crashes, or we say that the program crashes the Bit Machine. Crashing is distinct from halting. We will have a number of theorems that prove that a Bit Machine interpreting a Simplicity expression from a suitable initial state never crashes the Bit Machine; however in some of these cases the program may cause the Bit Machine to legitimately enter the halted state.

3.5.3 Executing Simplicity

We recursively translate a core Simplicity program, $t : A \vdash B$, into a program for the Bit Machine, $\langle\!\langle t \rangle\!\rangle$, called the naive translation:

$$\begin{aligned}
\langle\!\langle \mathsf{iden}_A \rangle\!\rangle &:= \mathrm{copy}(\mathrm{bitSize}(A)) \\
\langle\!\langle \mathsf{comp}_{A,B,C}\, s\, t \rangle\!\rangle &:= \mathrm{newFrame}(\mathrm{bitSize}(B)) \\
&\;;\; \langle\!\langle s \rangle\!\rangle \\
&\;;\; \mathrm{moveFrame} \\
&\;;\; \langle\!\langle t \rangle\!\rangle \\
&\;;\; \mathrm{dropFrame} \\
\langle\!\langle \mathsf{unit}_A \rangle\!\rangle &:= \mathrm{nop} \\
\langle\!\langle \mathsf{injl}_{A,B,C}\, t \rangle\!\rangle &:= \mathrm{write}(0); \mathrm{skip}(\mathrm{padL}(B,C)); \langle\!\langle t \rangle\!\rangle \\
\langle\!\langle \mathsf{injr}_{A,B,C}\, t \rangle\!\rangle &:= \mathrm{write}(1); \mathrm{skip}(\mathrm{padR}(B,C)); \langle\!\langle t \rangle\!\rangle \\
\langle\!\langle \mathsf{case}_{A,B,C,D}\, s\, t \rangle\!\rangle &:= (1 + \mathrm{padL}(A,B)) \star \langle\!\langle s \rangle\!\rangle \\
&\;\|\; (1 + \mathrm{padR}(A,B)) \star \langle\!\langle t \rangle\!\rangle \\
\langle\!\langle \mathsf{pair}_{A,B,C}\, s\, t \rangle\!\rangle &:= \langle\!\langle s \rangle\!\rangle ; \langle\!\langle t \rangle\!\rangle \\
\langle\!\langle \mathsf{take}_{A,B,C}\, t \rangle\!\rangle &:= \langle\!\langle t \rangle\!\rangle \\
\langle\!\langle \mathsf{drop}_{A,B,C}\, t \rangle\!\rangle &:= \mathrm{bitSize}(A) \star \langle\!\langle t \rangle\!\rangle
\end{aligned}$$

Theorem 3.7. *Given a well-typed core Simplicity program $t : A \vdash B$ and an input $a : A$, then*

$$[\Theta \rhd r_0 \wedge \ulcorner a \urcorner \cdot r_0' | w_0 \wedge \texttt{[?]}^{\mathrm{bitSize}(B)+m} \lhd \Xi] \rightarrowtail \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [\Theta \rhd r_0 \wedge \ulcorner a \urcorner \cdot r_0' | w_0 \cdot \ulcorner [\![t]\!](a) \urcorner \wedge \texttt{[?]}^{m} \lhd \Xi]$$

for any cell arrays r_0, r_0', w_0, any stacks Θ, Ξ, and any natural number m.

In particular, for a well-typed core Simplicity program $t : A \vdash B$, we have

$$[\wedge \ulcorner a \urcorner | \wedge \texttt{[?]}^{\mathrm{bitSize}(B)}] \rightarrowtail \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [\wedge \ulcorner a \urcorner | \ulcorner [\![t]\!](a) \urcorner \wedge]$$

which means we if we start the Bit Machine with only the input represented on the read stack, and enough space for the output on the write stack, the Bit Machine will compute the representation of the value $[\![t]\!](a)$ without crashing.

3.5.3.1 Tail Composition Optimisation (TCO)

Traditional imperative language implementations often make use of tail call optimization that occurs when the last command of a procedure is a call to a second procedure. Normally the first procedure's stack frame would be free after the second procedure returns. The tail call optimization instead frees the first procedure's stack frame prior to the call to the second procedure instead. This can reduce the overall memory use of the program.

The composition combinator, comp, in Simplicity plays a role similar to a procedure call. We can perform a tail composition optimization that moves the dropFrame instruction earlier to reduce the overall memory requirements needed to evaluate Simplicity programs. We define an alternate translation of Simplicity programs to Bit Machine programs via two mutually recursively defined functions, $\langle\!\langle\cdot\rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}}$ and $\langle\!\langle\cdot\rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}}$:

$$
\begin{aligned}
\langle\!\langle \mathsf{iden}_A \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} &:= \mathrm{copy}(\mathrm{bitSize}(A)) \\
\langle\!\langle \mathsf{comp}_{A,B,C}\, s\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} &:= \mathrm{newFrame}(\mathrm{bitSize}(B)) \\
&\quad ; \langle\!\langle s \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \\
&\quad ; \mathrm{moveFrame} \\
&\quad ; \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
\langle\!\langle \mathsf{unit}_A \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} &:= \mathrm{nop} \\
\langle\!\langle \mathsf{injl}_{A,B,C}\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} &:= \mathrm{write}(0); \mathrm{skip}(\mathrm{padL}(B,C)); \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \\
\langle\!\langle \mathsf{injr}_{A,B,C}\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} &:= \mathrm{write}(1); \mathrm{skip}(\mathrm{padR}(B,C)); \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \\
\langle\!\langle \mathsf{case}_{A,B,C,D}\, s\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} &:= (1 + \mathrm{padL}(A,B)) \star \langle\!\langle s \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \\
&\quad \mid\mid (1 + \mathrm{padR}(A,B)) \star \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \\
\langle\!\langle \mathsf{pair}_{A,B,C}\, s\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} &:= \langle\!\langle s \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}}; \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \\
\langle\!\langle \mathsf{take}_{A,B,C}\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} &:= \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \\
\langle\!\langle \mathsf{drop}_{A,B,C}\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} &:= \mathrm{bitSize}(A) \star \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}}
\end{aligned}
$$

$$
\begin{aligned}
\langle\!\langle \mathsf{iden}_A \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} &:= \mathrm{copy}(\mathrm{bitSize}(A)) \\
&\quad ; \mathrm{dropFrame} \\
\langle\!\langle \mathsf{comp}_{A,B,C}\, s\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} &:= \mathrm{newFrame}(\mathrm{bitSize}(B)) \\
&\quad ; \langle\!\langle s \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
&\quad ; \mathrm{moveFrame} \\
&\quad ; \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
\langle\!\langle \mathsf{unit}_A \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} &:= \mathrm{dropFrame} \\
\langle\!\langle \mathsf{injl}_{A,B,C}\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} &:= \mathrm{write}(0); \mathrm{skip}(\mathrm{padL}(B,C)); \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
\langle\!\langle \mathsf{injr}_{A,B,C}\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} &:= \mathrm{write}(1); \mathrm{skip}(\mathrm{padR}(B,C)); \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
\langle\!\langle \mathsf{case}_{A,B,C,D}\, s\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} &:= \mathrm{fwd}(1 + \mathrm{padL}(A,B)); \langle\!\langle s \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
&\quad \mid\mid \mathrm{fwd}(1 + \mathrm{padR}(A,B)); \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
\langle\!\langle \mathsf{pair}_{A,B,C}\, s\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} &:= \langle\!\langle s \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}}; \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
\langle\!\langle \mathsf{take}_{A,B,C}\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} &:= \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
\langle\!\langle \mathsf{drop}_{A,B,C}\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} &:= \mathrm{fwd}(\mathrm{bitSize}(A)); \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}}
\end{aligned}
$$

The definition of the $\langle\!\langle\cdot\rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}}$ translation is very similar to the naive one, except the dropFrame instruction at the end of the translation of the composition combinator is replaced by having a recursive call to $\langle\!\langle\cdot\rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}}$ instead. The definition of $\langle\!\langle\cdot\rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}}$ puts the dropFrame instruction in the translations of iden and unit. The bwd instructions are removed from the translations of case and drop. Lastly notice that the first recursive call in the translation of pair is to $\langle\!\langle\cdot\rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}}$.

Theorem 3.8. *Given a well-typed core Simplicity program $t\colon A \vdash B$ and an input $a\colon A$, then*

$$[\Theta \rhd r_0 \wedge \ulcorner a \urcorner \cdot r_0' | w_0 \wedge [?]^{\,\mathrm{bitSize}(B)+m} \lhd \Xi] \twoheadleftarrow \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \twoheadrightarrow [\Theta \rhd r_0 \wedge \ulcorner a \urcorner \cdot r_0' | w_0 \cdot \ulcorner [\![t]\!](a) \urcorner \wedge [?]^{\,m} \lhd \Xi]$$

and

$$[\Theta \rhd r_1 \rhd r_0 \wedge \ulcorner a \urcorner \cdot r_0' | w_0 \wedge [?]^{\,\mathrm{bitSize}(B)+m} \lhd \Xi] \twoheadleftarrow \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \twoheadrightarrow [\Theta \rhd r_1 | w_0 \cdot \ulcorner [\![t]\!](a) \urcorner \wedge [?]^{\,m} \lhd \Xi]$$

for any cell arrays r_0, r_0', w_0, any frame r_1, any stacks Θ, Ξ, and any natural number m.

In particular, for a well-typed core Simplicity program $t\colon A \vdash B$, we have

$$[{}_\wedge\ulcorner a \urcorner|_\wedge \mathtt{[?]}^{\mathrm{bitSize}(B)}] \rightarrowtail \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \twoheadrightarrow [{}_\wedge\ulcorner a \urcorner|\ulcorner [\![t]\!](a) \urcorner_\wedge]$$

3.6 Static Analysis

Static analysis lets us quickly compute properties of expressions, without the need for exhaustively executing expressions on all possible inputs. We use static analysis in Simplicity to bound the computation costs in terms of time and space of Simplicity expressions for all their inputs. The analysis we do are linear in the size of the DAG representing the Simplicity expression, and typically the analysis runs much faster than the cost of evaluating an expression on a single input. We can do this because the intermediate results of static analysis can be shared where there are shared sub-expressions.

3.6.1 Space Resources

The primary source of memory resources used by the Bit Machine is the cells used by all the frames that make of the state of Bit Machine. A secondary source of memory resources used comes from the overhead of the frames themselves, which need to store their boundaries or sizes, and the position of their cursors. In our analysis we will make a simplifying assumption that these boundaries / sizes / positions values are all of constant size. This assumption holds when the Bit Machine is implemented on real hardware which has an upper bound on its addressable memory and there is a limit on the number of Cells that can be held anyways.

To bound these resources we perform a static analysis to compute an upper bound on the maximum number of cells needed when executing a Simplicity program on the Bit Machine for any input, and we compute an upper bound on the maximum number of frames needed as well.

3.6.1.1 Maximum Cell Count Bound

We define the cell count of a frame to be the length of its underlying cell array and the cell count of a Bit Machine state to be the sum of the cell counts of all its frames.

$$\begin{aligned}
\mathrm{cellCount}(\mathtt{[}c_1 \mathtt{\flat} \cdots \underline{c}_i \cdots \mathtt{\natural} c_n \mathtt{]}) &:= n \\
\mathrm{cellCount}([r_n \rhd \ldots \rhd r_0 | w_0 \lhd \ldots \lhd w_m]) &:= \sum_{i=0}^{n} \mathrm{cellCount}(r_i) + \sum_{j=0}^{m} \mathrm{cellCount}(w_j) \\
\mathrm{cellCount}(\boxtimes) &:= 0
\end{aligned}$$

We define the cells required by a program $S_0 \rightarrowtail p \twoheadrightarrow S_1$ as the maximum cell count over every intermediate state.

$$\frac{}{\mathrm{cellsReq}(S \rightarrowtail \mathrm{nop} \twoheadrightarrow S) := \mathrm{cellCount}(S)}$$

$$\frac{i\colon S_0 \rightsquigarrow S_1}{\mathrm{cellsReq}(S_0 \rightarrowtail i \twoheadrightarrow S_1) := \max\left(\mathrm{cellCount}(S_0), \mathrm{cellCount}(S_1)\right)}$$

$$\frac{S_0 \rightarrowtail k_0 \twoheadrightarrow S_1 \qquad S_1 \rightarrowtail k_1 \twoheadrightarrow S_2}{\mathrm{cellsReq}(S_0 \rightarrowtail k_0; k_1 \twoheadrightarrow S_2) := \max\left(\mathrm{cellsReq}(S_0 \rightarrowtail k_0 \twoheadrightarrow S_1), \mathrm{cellsReq}(S_1 \rightarrowtail k_1 \twoheadrightarrow S_2)\right)}$$

$$\frac{[\Theta \rhd r_0 \cdot \mathtt{[}\underline{0}\mathtt{]} \cdot r_0' | w_0 \lhd \Xi] \rightarrowtail k_0 \twoheadrightarrow S}{\mathrm{cellsReq}([\Theta \rhd r_0 \cdot \mathtt{[}\underline{0}\mathtt{]} \cdot r_0' | w_0 \lhd \Xi] \rightarrowtail k_0 || \, k_1 \twoheadrightarrow S) := \mathrm{cellsReq}([\Theta \rhd r_0 \cdot \mathtt{[}\underline{0}\mathtt{]} \cdot r_0' | w_0 \lhd \Xi] \rightarrowtail k_0 \twoheadrightarrow S)}$$

$$\frac{[\Theta \rhd r_0 \cdot \mathtt{[}\underline{1}\mathtt{]} \cdot r_0' | w_0 \lhd \Xi] \rightarrowtail k_1 \twoheadrightarrow S}{\mathrm{cellsReq}([\Theta \rhd r_0 \cdot \mathtt{[}\underline{1}\mathtt{]} \cdot r_0' | w_0 \lhd \Xi] \rightarrowtail k_0 || \, k_1 \twoheadrightarrow S) := \mathrm{cellsReq}([\Theta \rhd r_0 \cdot \mathtt{[}\underline{1}\mathtt{]} \cdot r_0' | w_0 \lhd \Xi] \rightarrowtail k_1 \twoheadrightarrow S)}$$

$$\frac{}{\mathrm{cellsReq}(\boxtimes \rightarrowtail k \twoheadrightarrow \boxtimes) := 0}$$

Note that when executing a Simplicity expression on the Bit Machine, the size of the state prior and after execution is identical. For naive translation of Simplicity to the Bit Machine, we can write a simple recursive function that bounds the number of additional Cells needed to evaluate a Simplicity expression beyond the size of the initial and final state.

$$\begin{aligned}
\text{extraCellsBound}(\mathsf{iden}_A) &:= 0\\
\text{extraCellsBound}(\mathsf{comp}_{A,B,C}\, s\, t) &:= \text{bitSize}(B) + \max\left(\text{extraCellsBound}(s), \text{extraCellsBound}(t)\right)\\
\text{extraCellsBound}(\mathsf{unit}_A) &:= 0\\
\text{extraCellsBound}(\mathsf{injl}_{A,B,C}\, t) &:= \text{extraCellsBound}(t)\\
\text{extraCellsBound}(\mathsf{injr}_{A,B,C}\, t) &:= \text{extraCellsBound}(t)\\
\text{extraCellsBound}(\mathsf{case}_{A,B,C,D}\, s\, t) &:= \max\left(\text{extraCellsBound}(s), \text{extraCellsBound}(t)\right)\\
\text{extraCellsBound}(\mathsf{pair}_{A,B,C}\, s\, t) &:= \max\left(\text{extraCellsBound}(s), \text{extraCellsBound}(t)\right)\\
\text{extraCellsBound}(\mathsf{take}_{A,B,C}\, t) &:= \text{extraCellsBound}(t)\\
\text{extraCellsBound}(\mathsf{drop}_{A,B,C}\, t) &:= \text{extraCellsBound}(t)
\end{aligned}$$

Lemma 3.9. *For any core Simplicity expression $t\colon A \vdash B$, such that*

$$[\Theta \rhd r_0 | w_0 \lhd \Xi] \twoheadrightarrow \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [\Theta' \rhd r'_0 | w'_0 \lhd \Xi']$$

we have that

1. $\text{cellCount}([\Theta \rhd r_0 | w_0 \lhd \Xi]) = \text{cellCount}([\Theta' \rhd r'_0 | w'_0 \lhd \Xi'])$
2. $\text{cellsReq}([\Theta \rhd r_0 | w_0 \lhd \Xi] \twoheadrightarrow \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [\Theta' \rhd r'_0 | w'_0 \lhd \Xi']) \leq$
$\text{cellCount}([\Theta \rhd r_0 | w_0 \lhd \Xi]) + \text{extraCellsBound}(t).$

In particular for $a\colon A$ and

$$[{}_\wedge\ulcorner a \urcorner_\wedge | {}_\wedge[\mathtt{?}]^{\,\text{bitSize}(B)}] \twoheadrightarrow \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [{}_\wedge\ulcorner a \urcorner_\wedge | \ulcorner [\![t]\!](a) \urcorner_\wedge]$$

we have that

$\text{cellsReq}([{}_\wedge\ulcorner a \urcorner_\wedge | {}_\wedge[\mathtt{?}]^{\,\text{bitSize}(B)}] \twoheadrightarrow \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [{}_\wedge\ulcorner a \urcorner_\wedge | \ulcorner [\![t]\!](a) \urcorner_\wedge]) \leq$
$\text{bitSize}(A) + \text{bitSize}(B) + \text{extraCellsBound}(t).$

We can compute a tighter bound for TCO translation, but the calculation is a bit more complicated. The number of extra cells needed depends on whether TCO is in the "on" state, and what the size of the active read frame is.

$$\begin{aligned}
\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(\mathsf{iden}_A)(r) &:= 0\\
\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(\mathsf{comp}_{A,B,C}\, s\, t)\,(r) &:= \text{bitSize}(B) + \max\big(\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(s)(r),\\
&\qquad \text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t)(\text{bitSize}(B)) - r\big)\\
\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(\mathsf{unit}_A)(r) &:= 0\\
\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(\mathsf{injl}_{A,B,C}\, t)(r) &:= \text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t)\\
\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(\mathsf{injr}_{A,B,C}\, t)(r) &:= \text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t)\\
\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(\mathsf{case}_{A,B,C,D}\, s\, t)(r) &:= \max\big(\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(s)(r),\\
&\qquad \text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t)(r)\big)\\
\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(\mathsf{pair}_{A,B,C}\, s\, t)(r) &:= \max\big(\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(s)(0),\\
&\qquad \text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t)(r)\big)\\
\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(\mathsf{take}_{A,B,C}\, t)(r) &:= \text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t)\\
\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(\mathsf{drop}_{A,B,C}\, t)(r) &:= \text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t)
\end{aligned}$$

Lemma 3.10. *For any core Simplicity expression $t\colon A \vdash B$, such that*

$$[\Theta_{\text{on}} \rhd r_{\text{on},0} | w_{\text{on},0} \lhd \Xi_{\text{on}}] \twoheadrightarrow \langle\!\langle t \rangle\!\rangle^{\text{TCO}}_{\text{on}} \twoheadrightarrow [\Theta'_{\text{on}} \rhd r'_{\text{on},0} | w'_{\text{on},0} \lhd \Xi'_{\text{on}}]$$

and

$$[\Theta_{\text{off}} \rhd r_{\text{off},0} | w_{\text{off},0} \lhd \Xi_{\text{off}}] \twoheadleftarrow \langle\!\langle t \rangle\!\rangle^{\text{TCO}}_{\text{off}} \twoheadrightarrow [\Theta'_{\text{off}} \rhd r'_{\text{off},0} | w'_{\text{off},0} \lhd \Xi'_{\text{off}}]$$

we have that

1. $\text{cellCount}([\Theta_{\text{on}} \rhd r_{\text{on},0} | w_{\text{on},0} \lhd \Xi_{\text{on}}]) = \text{cellCount}(r_{\text{on},0}) + \text{cellCount}([\Theta'_{\text{on}} \rhd r'_{\text{on},0} | w'_{\text{on},0} \lhd \Xi'_{\text{on}}])$ *and*
   $\text{cellCount}([\Theta_{\text{off}} \rhd r_{\text{off},0} | w_{\text{off},0} \lhd \Xi_{\text{off}}]) = \text{cellCount}([\Theta'_{\text{off}} \rhd r'_{\text{off},0} | w'_{\text{off},0} \lhd \Xi'_{\text{off}}])$
2. $\text{cellsReq}([\Theta_{\text{on}} \rhd r_{\text{on},0} | w_{\text{on},0} \lhd \Xi_{\text{on}}] \twoheadleftarrow \langle\!\langle t \rangle\!\rangle^{\text{TCO}}_{\text{on}} \twoheadrightarrow [\Theta'_{\text{on}} \rhd r'_{\text{on},0} | w'_{\text{on},0} \lhd \Xi'_{\text{on}}]) \leq$
   $\text{cellCount}([\Theta_{\text{on}} \rhd r_{\text{on},0} | w_{\text{on},0} \lhd \Xi_{\text{on}}]) + \text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t)(\text{cellCount}(r_{\text{on},0}))$ *and*
   $\text{cellsReq}([\Theta_{\text{off}} \rhd r_{\text{off},0} | w_{\text{off},0} \lhd \Xi_{\text{off}}] \twoheadleftarrow \langle\!\langle t \rangle\!\rangle^{\text{TCO}}_{\text{off}} \twoheadrightarrow [\Theta'_{\text{off}} \rhd r'_{\text{off},0} | w'_{\text{off},0} \lhd \Xi'_{\text{off}}]) \leq$
   $\text{cellCount}([\Theta_{\text{off}} \rhd r_{\text{off},0} | w_{\text{off},0} \lhd \Xi_{\text{off}}]) + \text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t)(0).$

In particular for $a\colon A$ *and*

$$[\text{⌜}a\text{⌝}|\texttt{[?]}^{\,\text{bitSize}(B)}] \twoheadleftarrow \langle\!\langle t \rangle\!\rangle^{\text{TCO}}_{\text{off}} \twoheadrightarrow [\text{⌜}a\text{⌝}|\text{⌜}[\![t]\!](a)\text{⌝}]$$

we have that

$$\text{cellsReq}([\text{⌜}a\text{⌝}|\texttt{[?]}^{\,\text{bitSize}(B)}] \twoheadleftarrow \langle\!\langle t \rangle\!\rangle^{\text{TCO}}_{\text{off}} \twoheadrightarrow [\text{⌜}a\text{⌝}|\text{⌜}[\![t]\!](a)\text{⌝}]) \leq \text{bitSize}(A) + \text{bitSize}(B) + \text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t)(0).$$

The problem with $\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t)$ is that it is effectively a dynamic analysis because its result is a function. We cannot directly use this definition to perform a static analysis because we cannot cache and reuse results on shared sub-expressions. Fortunately, we can characterize the set of possible functions returned by $\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}$ by a pair of parameters.

$$\text{interp}^{\text{TCO}}\langle n, m\rangle(r) := \max(n - r, m)$$

We can write a static analysis to compute the pair of parameters that characterize the results of $\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}$.

$$\begin{aligned}
\text{extraCellsBound}^{\text{TCO}}_{\text{static}}(\mathsf{iden}_A) &:= \langle 0, 0\rangle \\
\text{extraCellsBound}^{\text{TCO}}_{\text{static}}(\mathsf{comp}_{A,B,C}\, s\, t) &:= \langle \max(r_b + n_s, n_t, r_b + m_t), r_b + m_s\rangle \\
&\text{where } \langle n_s, m_s\rangle := \text{extraCellsBound}^{\text{TCO}}_{\text{static}}(s) \\
&\text{and } \langle n_t, m_t\rangle := \text{extraCellsBound}^{\text{TCO}}_{\text{static}}(t) \\
&\text{and } r_b := \text{bitSize}(B) \\
\text{extraCellsBound}^{\text{TCO}}_{\text{static}}(\mathsf{unit}_A) &:= \langle 0, 0\rangle \\
\text{extraCellsBound}^{\text{TCO}}_{\text{static}}(\mathsf{injl}_{A,B,C}\, t) &:= \text{extraCellsBound}^{\text{TCO}}_{\text{static}}(t) \\
\text{extraCellsBound}^{\text{TCO}}_{\text{static}}(\mathsf{injr}_{A,B,C}\, t) &:= \text{extraCellsBound}^{\text{TCO}}_{\text{static}}(t) \\
\text{extraCellsBound}^{\text{TCO}}_{\text{static}}(\mathsf{case}_{A,B,C,D}\, s\, t) &:= \langle \max(n_s, n_t), \max(m_s, m_t)\rangle \\
&\text{where } \langle n_s, m_s\rangle := \text{extraCellsBound}^{\text{TCO}}_{\text{static}}(s) \\
&\text{and } \langle n_t, m_t\rangle := \text{extraCellsBound}^{\text{TCO}}_{\text{static}}(t) \\
\text{extraCellsBound}^{\text{TCO}}_{\text{static}}(\mathsf{pair}_{A,B,C}\, s\, t) &:= \langle n_t, \max(n_s, m_s, m_t)\rangle \\
&\text{where } \langle n_s, m_s\rangle := \text{extraCellsBound}^{\text{TCO}}_{\text{static}}(s) \\
&\text{and } \langle n_t, m_t\rangle := \text{extraCellsBound}^{\text{TCO}}_{\text{static}}(t) \\
\text{extraCellsBound}^{\text{TCO}}_{\text{static}}(\mathsf{take}_{A,B,C}\, t) &:= \text{extraCellsBound}^{\text{TCO}}_{\text{static}}(t) \\
\text{extraCellsBound}^{\text{TCO}}_{\text{static}}(\mathsf{drop}_{A,B,C}\, t) &:= \text{extraCellsBound}^{\text{TCO}}_{\text{static}}(t)
\end{aligned}$$

When computing $\text{extraCellsBound}^{\text{TCO}}_{\text{static}}(t)$ resulting values for shared sub-expressions can be shared, making $\text{extraCellsBound}^{\text{TCO}}_{\text{static}}$ a static analysis. We can use $\text{interp}^{\text{TCO}}$ and $\text{extraCellsBound}^{\text{TCO}}_{\text{static}}$ to compute $\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}$ for our bound on cell count.

Lemma 3.11. $\text{extraCellsBound}^{\text{TCO}}_{\text{dyn}}(t) = \text{interp}^{\text{TCO}}(\text{extraCellsBound}^{\text{TCO}}_{\text{static}}(t))$.

Corollary 3.12. *For any core Simplicity expression* $t\colon A \vdash B$ *and* $a\colon A$ *such that*

$$[\text{⌜}a\text{⌝}|\texttt{[?]}^{\,\text{bitSize}(B)}] \twoheadleftarrow \langle\!\langle t \rangle\!\rangle^{\text{TCO}}_{\text{off}} \twoheadrightarrow [\text{⌜}a\text{⌝}|\text{⌜}[\![t]\!](a)\text{⌝}]$$

we have that

$\mathrm{cellsReq}([{}_\wedge\ulcorner a\urcorner|{}_\wedge\texttt{[?]}^{\mathrm{bitSize}(B)}] \twoheadrightarrow \langle\!\langle t\rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \twoheadrightarrow [{}_\wedge\ulcorner a\urcorner|\ulcorner[\![t]\!](a)\urcorner{}_\wedge]) \leq \mathrm{bitSize}(A) + \mathrm{bitSize}(B) + \max(n, m)$

where $\langle n, m\rangle := \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t)$.

3.6.1.2 Maximum Frame Count Bound

3.6.2 Time Resources

3.7 Commitment Merkle Root

In modern Bitcoin, users who use P2SH (pay to script hash) do not commit funds directly to Bitcoin Script, rather they commit to a hash of their Bitcoin Script. Only when they wish to redeem their funds do they reveal their Bitcoin Script for execution. Bitcoin's consensus protocol enforces that the Bitcoin Script presented during redemption has a hash that matches the committed hash.

Simplicity is designed to work in the same way. However, instead of a linear hash of a serialized Simplicity program (Section 7.2) we follow the tree structure of a Simplicity expression and compute a commitment Merkle root of its syntax tree. Below we define both the commitment Merkle root of a Simplicity expression $t : A \vdash B$ as $\#^c(t) : 2^{256}$ as well as a value $\#^i(t) : 2^{256}$ that will later be used to define a varianted for identity Merkle Roots (Section 7.2.3). Below $\alpha \in \{c, i\}$ and hence for core Simplicity expression t, $\#^c(t) = \#^i(t)$, but they will no longer be equal when we consider some extensions to core Simplicity.

$$
\begin{aligned}
\#^\alpha(\mathsf{iden}_A) &:= \mathrm{SHA256}^{\mathrm{tag}^c_{\mathsf{iden}}}_{\mathrm{IV}} \\
\#^\alpha(\mathsf{comp}_{A,B,C}\, s\, t) &:= \mathrm{SHA256}_{\mathrm{Block}}\langle \mathrm{SHA256}^{\mathrm{tag}^c_{\mathsf{comp}}}_{\mathrm{IV}}, \langle \#^\alpha(s), \#^\alpha(t)\rangle\rangle \\
\#^\alpha(\mathsf{unit}_A) &:= \mathrm{SHA256}^{\mathrm{tag}^c_{\mathsf{unit}}}_{\mathrm{IV}} \\
\#^\alpha(\mathsf{injl}_{A,B,C}\, t) &:= \mathrm{SHA256}_{\mathrm{Block}}\Big\langle \mathrm{SHA256}^{\mathrm{tag}^c_{\mathsf{injl}}}_{\mathrm{IV}}, \langle \lfloor 0\rfloor_{256}, \#^\alpha(t)\rangle\Big\rangle \\
\#^\alpha(\mathsf{injr}_{A,B,C}\, t) &:= \mathrm{SHA256}_{\mathrm{Block}}\langle \mathrm{SHA256}^{\mathrm{tag}^c_{\mathsf{injr}}}_{\mathrm{IV}}, \langle \lfloor 0\rfloor_{256}, \#^\alpha(t)\rangle\rangle \\
\#^\alpha(\mathsf{case}_{A,B,C,D}\, s\, t) &:= \mathrm{SHA256}_{\mathrm{Block}}\langle \mathrm{SHA256}^{\mathrm{tag}^c_{\mathsf{case}}}_{\mathrm{IV}}, \langle \#^\alpha(s), \#^\alpha(t)\rangle\rangle \\
\#^\alpha(\mathsf{pair}_{A,B,C}\, s\, t) &:= \mathrm{SHA256}_{\mathrm{Block}}\langle \mathrm{SHA256}^{\mathrm{tag}^c_{\mathsf{pair}}}_{\mathrm{IV}}, \langle \#^\alpha(s), \#^\alpha(t)\rangle\rangle \\
\#^\alpha(\mathsf{take}_{A,B,C}\, t) &:= \mathrm{SHA256}_{\mathrm{Block}}\langle \mathrm{SHA256}^{\mathrm{tag}^c_{\mathsf{take}}}_{\mathrm{IV}}, \langle \lfloor 0\rfloor_{256}, \#^\alpha(t)\rangle\rangle \\
\#^\alpha(\mathsf{drop}_{A,B,C}\, t) &:= \mathrm{SHA256}_{\mathrm{Block}}\Big\langle \mathrm{SHA256}^{\mathrm{tag}^c_{\mathsf{drop}}}_{\mathrm{IV}}, \langle \lfloor 0\rfloor_{256}, \#^\alpha(t)\rangle\Big\rangle
\end{aligned}
$$

Here we are directly using SHA-256's compression function, $\mathrm{SHA256}_{\mathrm{Block}}\langle i, b\rangle$, which takes two arguments. The first argument, i, is a 256-bit tagged initial value. The second value, b, is a 512-bit block of data. Above we divide a block into two 256-bit values, $\langle b_0, b_1\rangle$, and recursively pass Merkle roots into the compression function.

Like static analysis, the time needed to computing the commitment Merkle root is linear in the size of the DAG representing the term because the intermediate results on sub-expressions can be shared.

We define unique tags tag^c_x for every combinator.

$$
\begin{aligned}
\mathrm{tag}^c_{\mathsf{iden}} &:= \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f6964656e]}_{2^8} \\
\mathrm{tag}^c_{\mathsf{comp}} &:= \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f636f6d70]}_{2^8} \\
\mathrm{tag}^c_{\mathsf{unit}} &:= \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f756e6974]}_{2^8} \\
\mathrm{tag}^c_{\mathsf{injl}} &:= \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f696e6a6c]}_{2^8} \\
\mathrm{tag}^c_{\mathsf{injr}} &:= \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f696e6a72]}_{2^8} \\
\mathrm{tag}^c_{\mathsf{case}} &:= \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f63617365]}_{2^8} \\
\mathrm{tag}^c_{\mathsf{pair}} &:= \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f70616972]}_{2^8} \\
\mathrm{tag}^c_{\mathsf{take}} &:= \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f74616b65]}_{2^8} \\
\mathrm{tag}^c_{\mathsf{drop}} &:= \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f64726f70]}_{2^8}
\end{aligned}
$$

Notice that the type annotations for expressions are not included in the commitment Merkle root. We will rely on type inference to derive principle type annotations (see Section 7.1.1). Later, we will make use of this flexibility when pruning unused branches from case expressions (see Section 4.3.2.1).

3.8 Type Merkle Root

We also define a Merkle root that follows the tree sturcture of types in the same way that we defined the commitment Merkle Root.

$$\begin{aligned}
\#^{\mathrm{ty}}(\mathbb{1}) &:= \mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^{\mathrm{ty}}_{\mathsf{one}}} \\
\#^{\mathrm{ty}}(A+B) &:= \mathrm{SHA256}_{\mathrm{Block}}\left\langle \mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^{\mathrm{ty}}_{\mathsf{sum}}}, \langle \#^{\mathrm{ty}}(A), \#^{\mathrm{ty}}(B)\rangle \right\rangle \\
\#^{\mathrm{ty}}(A\times B) &:= \mathrm{SHA256}_{\mathrm{Block}}\left\langle \mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^{\mathrm{ty}}_{\mathsf{prod}}}, \langle \#^{\mathrm{ty}}(A), \#^{\mathrm{ty}}(B)\rangle \right\rangle
\end{aligned}$$

We define unique tags $\mathrm{tag}^{\mathrm{ty}}_x$ for every sort of type.

$$\begin{aligned}
\mathrm{tag}^{\mathrm{ty}}_{\mathsf{one}} &:= \texttt{[53696d706c69636974792d44726166741f547970651f756e6974]}_{2^8} \\
\mathrm{tag}^{\mathrm{ty}}_{\mathsf{sum}} &:= \texttt{[53696d706c69636974792d44726166741f547970651f73756d]}_{2^8} \\
\mathrm{tag}^{\mathrm{ty}}_{\mathsf{prod}} &:= \texttt{[53696d706c69636974792d44726166741f547970651f70726f64]}_{2^8}
\end{aligned}$$

Chapter 4
Simplicity Extensions

The core Simplicity completeness theorem (Theorem 3.3) proves that the core Simplicity language is already computationally complete for Simplicity types. Our primary method of extending Simplicity is by adding expressions with side-effects. We will use monads to formally specify these new effects.

4.1 Monadic Semantics

We define a new interpretation of Simplicity expressions, $t : A \vdash B$, whose denotations are Kleisli morphisms, $[\![t]\!]^{\mathcal{M}} : A \to \mathcal{M}\, B$.

$$\begin{aligned}
[\![\mathsf{iden}_A]\!]^{\mathcal{M}}(a) &:= \eta^{\mathcal{M}}(a)\\
[\![\mathsf{comp}_{A,B,C}\, s\, t]\!]^{\mathcal{M}}(a) &:= ([\![t]\!]^{\mathcal{M}} \leftarrowtail [\![s]\!]^{\mathcal{M}})(a)\\
[\![\mathsf{unit}_A]\!]^{\mathcal{M}}(a) &:= \eta^{\mathcal{M}}\langle\rangle\\
[\![\mathsf{injl}_{A,B,C}\, t]\!]^{\mathcal{M}}(a) &:= \mathcal{M}\sigma^{\mathbf{L}}([\![t]\!]^{\mathcal{M}}(a))\\
[\![\mathsf{injr}_{A,B,C}\, t]\!]^{\mathcal{M}}(a) &:= \mathcal{M}\sigma^{\mathbf{R}}([\![t]\!]^{\mathcal{M}}(a))\\
[\![\mathsf{case}_{A,B,C,D}\, s\, t]\!]^{\mathcal{M}}\langle\sigma^{\mathbf{L}}(a), c\rangle &:= [\![s]\!]^{\mathcal{M}}\langle a, c\rangle\\
[\![\mathsf{case}_{A,B,C,D}\, s\, t]\!]^{\mathcal{M}}\langle\sigma^{\mathbf{R}}(b), c\rangle &:= [\![t]\!]^{\mathcal{M}}\langle b, c\rangle\\
[\![\mathsf{pair}_{A,B,C}\, s\, t]\!]^{\mathcal{M}}(a) &:= \phi^{\mathcal{M}}\langle [\![s]\!]^{\mathcal{M}}(a), [\![t]\!]^{\mathcal{M}}(a)\rangle\\
[\![\mathsf{take}_{A,B,C}\, t]\!]^{\mathcal{M}}\langle a, b\rangle &:= [\![t]\!]^{\mathcal{M}}(a)\\
[\![\mathsf{drop}_{A,B,C}\, t]\!]^{\mathcal{M}}\langle a, b\rangle &:= [\![t]\!]^{\mathcal{M}}(b)
\end{aligned}$$

The above interpretation for Kleisli morphisms is nearly uniquely defined (under the requirement of parametericity). Many well-typed variations of the definition above end up being equivalent due to the monad laws. The main choice we have is between using $\phi^{\mathcal{M}}_{A,B}$ or $\bar{\phi}^{\mathcal{M}}_{A,B}$ in the definition of $[\![\mathsf{pair}\, s\, t]\!]^{\mathcal{M}}$. The only other definitions amount to duplicating the effects of sub-expressions.

To ensure that all these possible choices are immaterial, we demand that $\mathcal{M}$ be a commutative, idempotent monad when interpreting Simplicity expressions. This lets us ignore the order of effects, and duplication of effects, which simplifies reasoning about Simplicity programs. It also provides an opportunity for a Simplicity optimizer to, for example, reorder pairs without worrying about changing the denotational semantics.

Theorem 4.1. *For any core Simplicity expression, $t : A \vdash B$, we have $[\![t]\!]^{\mathcal{M}} := \eta^{\mathcal{M}}_B \circ [\![t]\!]$.*

Corollary 4.2. *For any core Simplicity expression, $t : A \vdash B$, we have $[\![t]\!]^{\mathrm{Id}} := [\![t]\!]$.*

Notice that our monadic semantics are strict in their side-effects in the sense that the definition of $[\![\mathsf{pair}\, s\, t]\!]^{\mathcal{M}}$ implies that the side-effects of $[\![s]\!]^{\mathcal{M}}$ and $[\![t]\!]^{\mathcal{M}}$ are both realized even if it ends up that one (or both) of the values returned by s and t end up never used.

4.2 Witness

Our first extension to core Simplicity is the witness expression. The language that uses this extension is called *Simplicity with witnesses*.

$$\frac{b : B}{\mathsf{witness}_{A,B}\, b : A \vdash B}$$

The denotational semantics of the witness expression is simply a constant function that returns its parameter.

$$[\![\mathsf{witness}_{A,B}\, b]\!]^{\mathcal{M}}(a) := \eta_B^{\mathcal{M}}(b)$$

As far as semantics goes, this extension does not provide any new expressivity. A constant function for any value b can already be expressed in core Simplicity using $\mathsf{scribe}_{A,B}(b)$. The difference between scribe and witness expressions lies in their commitment Merkle root.

$$\#^c(\mathsf{witness}_{A,B}\, b) \;:=\; \mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^c_{\mathsf{witness}}}$$

where $\mathrm{tag}^c_{\mathsf{witness}}$ value is a unique value.

$$\mathrm{tag}^c_{\mathsf{witness}} \;:=\; \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f7769746e657373]}_{2^8}$$

Notice that a witness b expression does not commit to its parameter in the commitment root. This means that at redemption time a witness expression's parameter, called a *witness value*, could be set to any value. This differs from the identity root (Section 7.2.3) which will include a commitment to its parameter.

Witness values play the same role as Bitcoin Script's input stack in its `sigScript` or Segwit's `witness`. They act as inputs to Simplicity programs. Rather than accepting arguments as inputs and passing them down to where they are needed, witness expressions lets input data appear right where it is needed.

4.2.1 Elided Computation

4.2.2 Type Inference with Witness

Like other expressions, a witness expression does not commit to its type in its commitment Merkle root. Type inference is used to compute the minimal type needed for each witness expression (see Section 7.1.1) This helps ensures that third parties cannot perform witness malleation to add unused data on transactions during transit.

4.3 Assertions and Failure

Our first side-effect will be aborting a computation. New assertion and fail expressions make use of this effect. The language that uses this extension is called *Simplicity with assertions*.

$$\frac{s : A \times C \vdash D \qquad h : 2^{256}}{\mathsf{assertl}_{A,B,C,D}\, s\, h : (A + B) \times C \vdash D}$$

$$\frac{h : 2^{256} \qquad t : B \times C \vdash D}{\mathsf{assertr}_{A,B,C,D}\, h\, t : (A + B) \times C \vdash D}$$

$$\frac{h : 2^{512}}{\mathsf{fail}_{A,B}\, h : A \vdash B}$$

Assertions serve a dual purpose. One purpose is to replicate the behaviour of Bitcoin Script's `OP_VERIFY` and similar operations that are used to validate checks on programmable conditions, such as verifying that a digital signature verification passes or causing the program to abort otherwise.

The second purpose is to support pruning of unused case branches during redemption. The 256-bit value is used in the commitment Merkle root computation to hold Merkle root of the pruned branches. This will be covered in Section 4.3.2.1.

Because we are extending Simplicity's semantics to support an abort effect, there is no harm in adding a generic fail expression. The parameter to the fail expression is used to support salted expressions (see Section 4.3.2.2). We will see that fail expressions never manifest themselves within a blockchain's consensus protocol.

4.3.1 Denotational Semantics

Given an commutative, idempotent monad with zero, $\mathcal{M}$, we extend the monadic semantics for Simplicity expressions, $[\![t]\!]^{\mathcal{M}}$, to include assertion expressions:

$$\begin{aligned}
[\![\mathsf{assertl}_{A,B,C,D}\, s\, h]\!]^{\mathcal{M}}\langle \sigma^{\mathbf{L}}(a), c\rangle &:= [\![s]\!]^{\mathcal{M}}\langle a, c\rangle\\
[\![\mathsf{assertl}_{A,B,C,D}\, s\, h]\!]^{\mathcal{M}}\langle \sigma^{\mathbf{R}}(b), c\rangle &:= \emptyset_D^{\mathcal{M}}\\
[\![\mathsf{assertr}_{A,B,C,D}\, h\, t]\!]^{\mathcal{M}}\langle \sigma^{\mathbf{L}}(a), c\rangle &:= \emptyset_D^{\mathcal{M}}\\
[\![\mathsf{assertr}_{A,B,C,D}\, h\, t]\!]^{\mathcal{M}}\langle \sigma^{\mathbf{R}}(b), c\rangle &:= [\![t]\!]^{\mathcal{M}}\langle b, c\rangle\\
[\![\mathsf{fail}_{A,B}\, h]\!]^{\mathcal{M}}(a) &:= \emptyset_B^{\mathcal{M}}
\end{aligned}$$

Notice that the h parameters are ignored in the semantics. They will be used instead for the Merkle root definitions in Section 4.3.2.

A term in the language of core Simplicity extended with witnesses and assertions, $t : A \vdash B$, can be interpreted as a function returning an optional result: $[\![t]\!]^{\mathrm{S}} : A \to \mathrm{S}B$, using the option monad (see Section 2.3.4.1).

Theorem 4.3. *For any core Simplicity expression with assertions, $t : A \vdash B$, and any commutative idempotent monad with zero $\mathcal{M}$, we have $[\![t]\!]^{\mathcal{M}} := \iota^{\mathcal{M}}_{\mathrm{S},B} \circ [\![t]\!]^{\mathrm{S}}$.*

4.3.2 Merkle Roots

We extend the definition of commitment Merkle root and identity Merkle root to support the new assertion and fail expressions

$$\begin{aligned}
\#^{\alpha}(\mathsf{assertl}_{A,B,C,D}\, s\, h) &:= \mathrm{SHA256}_{\mathrm{Block}}\langle \mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^c_{\mathsf{case}}}, \langle \#^{\alpha}(s), h\rangle\rangle\\
\#^{\alpha}(\mathsf{assertr}_{A,B,C,D}\, h\, t) &:= \mathrm{SHA256}_{\mathrm{Block}}\langle \mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^c_{\mathsf{case}}}, \langle h, \#^{\alpha}(t)\rangle\rangle\\
\#^{\alpha}(\mathsf{fail}_{A,B}\, h) &:= \mathrm{SHA256}_{\mathrm{Block}}\big\langle \mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^c_{\mathsf{fail}}}, h\big\rangle
\end{aligned}$$

where $\mathrm{tag}^c_{\mathsf{fail}}$ value is a unique value.

$$\mathrm{tag}^c_{\mathsf{fail}} := \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f6661696c]}_{2^8}$$

It is important to notice that we are reusing $\mathrm{tag}^c_{\mathsf{case}}$ when tagging assertions in their commitment Merkle root. Also notice that the h value, which was ignored in the semantics, is used in the commitment Merkle root. Together this allows an assertion expression to substitute for a case expression at redemption time while maintaining the same commitment Merkle root. This enables a feature of Simplicity called pruning.

4.3.2.1 Pruning Unused case Branches

The commitment Merkle roots of the assertion expression reuses $\mathrm{tag}^c_{\mathsf{case}}$ in the compression function. This means that the following identities hold.

$$\#^c(\mathsf{assertl}_{A,B,C,D}\, s\, \#^c(t)) = \#^c(\mathsf{case}_{A,B,C,D}\, s\, t) = \#^c(\mathsf{assertr}_{A,B,C,D}\, \#^c(s)\, t)$$

In particular, it means that when a case expression is used at commitment time, it can be replaced by an assertion expression. If we substitute a case with an assertion expression and that assertion fails during evaluation, then the whole transaction will be deemed invalid and rejected. On the other hand if the assertion does not fail, then we are guaranteed to end up with the same result as before (which ultimately could still be failure due to a later assertion failure). Therefore, assuming the transaction is valid, a substitution of assertions will not change the semantics.

We can take advantage of this by performing this substitution at redemption time. We can effectively replace any unused branch in a case expression with its commitment Merkle root. In fact, we will require this replacement to occur during redemption (see Section ⟨reference|TODO⟩).

For those cases where we want to use an assertion at commitment time, for example when performing something similar to Bitcoin Script's `OP_VERIFY`, we use the following derived assert combinator,

$$\frac{t : A \vdash 2}{\mathsf{assert}_A\, t := t \vartriangle \mathsf{unit}; \mathsf{assertr}\, \#^c(\mathsf{fail}\, \lfloor 0 \rfloor_{512})\, \mathsf{unit} : A \vdash \mathbb{1}}$$

where $\lfloor 0 \rfloor_{512}$ is used as a canonical parameter for fail. Naturally, the $\lfloor 0 \rfloor_{512}$ parameter can be replaced with any value. This can be used as a method of salting expression, which is the subject of the next section.

Theorem 4.4. *For any core Simplicity expression* $t : A \vdash 2$,

$$[\![\mathsf{assert}_A\, t]\!]^S = [\![t]\!].$$

4.3.2.2 Salted Expressions

During pruning, unused branches are replaced by its commitment Merkle root. Since hashes are one way functions, one might believe that third parties will be unable to recover the pruned branch from just its commitment Merkle root. However, this argument is not so straightforward. Whether or not the expression can be recovered from just its commitment Merkle root depends on how much entropy the pruned expression contains. Third parties can grind, testing many different expressions, until they find one whose commitment Merkle root matches the one occurring in the assertion. If the entropy of the pruned expression is low, then this grinding is feasible.

Some expressions naturally have high entropy. For example, any branch that contains a commitment to a public key will have at least the entropy of the public key space. However, this only holds so long as that public key is not reused nor will ever be reused elsewhere.

For expressions that reuse public keys, or otherwise naturally having low entropy, one can add salt, which is random data, to increase its entropy. There are several possible ways to incorporate random data into a Simplicity expression without altering the program's semantics. One way is to incorporate the fail expression which lets us directly incorporate random into is commitment Merkle root.

Given a block of random data, h: 2^{512}, and a Simplicity expression $t : A \vdash B$, we can define two salted variants of t:

$$\begin{aligned} \mathsf{salted}^0\, h\, t &:= \mathsf{witness}\, 0_2 \vartriangle \mathsf{iden}; \mathsf{assertl}\, (\mathsf{drop}\, t)\, \#^c(\mathsf{fail}\, h) : A \vdash B \\ \mathsf{salted}^1\, h\, t &:= \mathsf{witness}\, 1_2 \vartriangle \mathsf{iden}; \mathsf{assertr}\, \#^c(\mathsf{fail}\, h)\, (\mathsf{drop}\, t) : A \vdash B \end{aligned}$$

The $\mathsf{salted}^b\, h\, t$ expression will have high entropy so long as the random data h has high entropy. By randomly choosing between these two variants, this method of salting obscures the fact that the expression is salted at all. Without knowing h, it is impractical to determine if $\#^c(\mathsf{fail}\, h)$ is the commitment Merkle root of a fail expression, or if it is a some other, high-entropy, alternate expression that the redeemer has simply chosen not to execute.

By explicitly using the fail expression here, one has the option prove that these alternative branches are unexecutable by revealing the value h. If the Simplicity expression is part of a multi-party smart contract, it maybe required to reveal h (or prove in a deniable way that such an h exists) to all party members so everyone can vet the security properties of the overall smart contract.

Of course, lots of variations of this salted expression are possible.

4.4 Blockchain Primitives

We extend Simplicity with primitive expressions that provide blockchain specific features. Naturally the specifics of these primitive expressions depends on the specific blockchain application, but generally speaking the primitives allow reading data from the context that a Simplicity program is being executed within. This is usually the data of the encompassing transaction including details about the inputs and outputs of the transaction, and which specific input is being evaluated.

A blockchain application needs to provide a set of typed primitive expressions and a monad to capture the side-effects for these primitives. This monad should be a commutative, idempotent monad with zero in order to interpret Simplicity and its extensions. All primitive expressions must be monomorphic and have no parameters (i.e. they are not themselves combinators).

In the next section we will be detailing the primitives used for Bitcoin, or a Bitcoin-like application. In Appendix A we describe the primitives used for the Elements sidechain.

4.4.1 Bitcoin Transactions

For the Bitcoin application, Simplicity's primitives will be primarily focuses on accessing the *signed transaction data*, which is the data that is hashed and signed in Bitcoin.

We define a record type that captures this environment, called BCEnv.

$$
\begin{aligned}
\text{Lock} &:= 2^{32}\\
\text{Value} &:= 2^{64}\\
\text{Outpoint} &:= 2^{256} \times 2^{32}\\
\text{SigInput} &:= \left\{ \begin{array}{l} \text{prevOutpoint} : \text{Outpoint}\\ \text{value} : \text{Value}\text{make this SigOutput? just add the segwitVerison/outputKey?}\\ \text{sequence} : 2^{32}\\ \text{annex} : \text{S}((2^8)^*) \text{ (excludes the } \mathtt{50}_{2^8} \text{ prefix)}\\ \text{scriptSig} : (2^8)^* \end{array} \right\}\\
\text{SigOutput} &:= \left\{ \begin{array}{l} \text{value} : \text{Value}\\ \text{scriptPubKey} : (2^8)^* \end{array} \right\}\\
\text{SigTx} &:= \left\{ \begin{array}{l} \text{version} : 2^{32}\\ \text{inputs} : \text{SigInput}^+\\ \text{outputs} : \text{SigOutput}^+\\ \text{lockTime} : \text{Lock} \end{array} \right\}\\
\text{TapEnv} &:= \left\{ \begin{array}{l} \text{leafVersion} : 2^8\\ \text{internalKey} : \text{PubKey}\\ \text{branch} : (2^{256})^* \end{array} \right\}\\
\text{BCEnv} &:= \left\{ \begin{array}{l} \text{tx} : \text{SigTx}\\ \text{ix} : 2^{32}\\ \text{tapEnv} : \text{TapEnv}\\ \text{scriptCMR} : 2^{256} \end{array} \right\}
\end{aligned}
$$

The type SigTx contains the signed transaction data. Following a design similar to BIP 143, this signed transaction data excludes transaction inputs' ScriptSigs and includes inputs' Bitcoin values. The ix field is input index whose redemption is being processed by this Simplicity program. The scriptCMR field holds the commitment Merkle root of the Simplicity program being executed.

The SigTx type given above allows for an unbounded number of inputs and outputs. However, there are limits imposed by the Bitcoin protocol. The number of inputs and outputs are limited to less than or equal to 2^{25} by Bitcoin's deserialization implementation. Similarly, the length of SigOutput's pubScript is limited to less than or equal to 2^{25} bytes. We assume all transactions to adhere to these limits when reasoning about Bitcoin transactions.

Furthermore, we assume that for every $e : \text{BCEnv}$ that $\lceil e[\text{ix}] \rceil < |e[\text{tx}][\text{inputs}]|$ so that "current" index being validated is, in fact, an input of the transaction. We also assume that the annex length, when it exists, is strictly less than 2^{25} (becaues it excludes the 0x50 prefix), $\lceil e[\text{tapEnv}][\text{leafVersion}] \rceil$ is even, and $|e[\text{tapEnv}][\text{branch}]| \leq 128$ in accordance with Taproot limitations.

Bitcoin's money supply is capped below $21\,000\,000 \times 10^8$ satoshi, therefore it is safe to assume that all monetary values are within this bound. In particular, we assume that for every $e : \mathrm{BCEnv}$ that the following inequalities hold.

$$0 \leq \mathrm{fold}_{\mathbb{N}}^{\langle +,0\rangle}((\lambda o.\, \lceil o[\mathrm{value}] \rceil_{64})^{+}(e[\mathrm{tx}][\mathrm{outputs}])) \leq \mathrm{fold}_{\mathbb{N}}^{\langle +,0\rangle}((\lambda i.\, \lceil i[\mathrm{value}] \rceil_{64})^{+}(e[\mathrm{tx}][\mathrm{inputs}])) \leq 21\,000\,000 \times 10^8$$

The monad we use for the Bitcoin application provides an environment effect (also known as a reader effect) that allows read-access to the BCEnv value defining the Simplicity program's evaluation context. We call this monad BC.

$$\begin{array}{rcl} \mathrm{BC}A & := & \mathrm{BCEnv} \to \mathrm{S}A \\ \mathrm{BC}f(a) & := & \lambda e : \mathrm{BCEnv}.\, \mathrm{S}f(a(e)) \end{array}$$

BC is a commutative, idempotent monad with zero:

$$\begin{array}{rcl} \eta_A^{\mathrm{BC}}(a) & := & \lambda e : \mathrm{BCEnv}.\, \eta_A^{\mathrm{S}}(a) \\ \mu_A^{\mathrm{BC}}(a) & := & \lambda e : \mathrm{BCEnv}.\, \mu_A^{\mathrm{S}}(\mathrm{S}(\lambda f.\, f(e))(a(e))) \\ \emptyset_A^{\mathrm{BC}} & := & \lambda e : \mathrm{BCEnv}.\, \emptyset_A^{\mathrm{S}} \end{array}$$

We define several new primitive expressions for reading data from a BCEnv value. The language that uses this extension is called *Simplicity with Bitcoin*.

$$\frac{}{\mathsf{version} : \mathbb{1} \vdash 2^{32}}$$

$$\frac{}{\mathsf{lockTime} : \mathbb{1} \vdash \mathrm{Lock}}$$

$$\frac{}{\mathsf{totalInputValue} : \mathbb{1} \vdash \mathrm{Value}}$$

$$\frac{}{\mathsf{currentIndex} : \mathbb{1} \vdash 2^{32}}$$

$$\frac{}{\mathsf{inputPrevOutpoint} : 2^{32} \vdash \mathrm{S}(\mathrm{Outpoint})}$$

$$\frac{}{\mathsf{inputValue} : 2^{32} \vdash \mathrm{S}(\mathrm{Value})}$$

$$\frac{}{\mathsf{inputSequence} : 2^{32} \vdash \mathrm{S}(2^{32})}$$

$$\frac{}{\mathsf{inputAnnexHash} : 2^{32} \vdash \mathrm{S}(\mathrm{S}(2^{256}))}$$

$$\frac{}{\mathsf{inputScriptSigHash} : 2^{32} \vdash \mathrm{S}(2^{256})}$$

$$\frac{}{\mathsf{totalOutputValue} : \mathbb{1} \vdash \mathrm{Value}}$$

$$\frac{}{\mathsf{outputValue} : 2^{32} \vdash \mathrm{S}(\mathrm{Value})}$$

$$\frac{}{\mathsf{outputScriptHash} : 2^{32} \vdash \mathrm{S}(2^{256})}$$

$$\frac{}{\mathsf{tapleafVersion} : \mathbb{1} \vdash 2^{8}}$$

$$\frac{}{\mathsf{tapbranch} : 2^{8} \vdash \mathrm{S}(2^{256})}$$

$$\frac{}{\mathsf{internalKey} : \mathbb{1} \vdash \mathrm{PubKey}}$$

$$\frac{}{\mathsf{scriptCMR} : \mathbb{1} \vdash 2^{256}}$$

4.4.1.1 Denotational Semantics

We extend the formal semantics of these new expressions as follows.

$$
\begin{aligned}
[\![\mathsf{version}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{tx}][\mathrm{version}]) \\
[\![\mathsf{lockTime}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{tx}][\mathrm{lockTime}]) \\
[\![\mathsf{totalInputValue}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(\mathrm{fold}^{\lfloor+\rfloor_{64}}((\lambda l.\, l[\mathrm{value}])^{+}(e[\mathrm{tx}][\mathrm{inputs}])))) \\
[\![\mathsf{currentIndex}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{ix}]) \\
[\![\mathsf{inputPrevOutpoint}]\!]^{\mathrm{BC}}(i) &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, l[\mathrm{prevOutpoint}])(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{inputValue}]\!]^{\mathrm{BC}}(i) &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, l[\mathrm{value}])(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{inputSequence}]\!]^{\mathrm{BC}}(i) &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, l[\mathrm{sequence}])(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{inputAnnexHash}]\!]^{\mathrm{BC}}(i) &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, \mathrm{S}(\mathrm{SHA256})(l[\mathrm{annex}]))(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{inputScriptSigHash}]\!]^{\mathrm{BC}}(i) &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, \mathrm{SHA256}(l[\mathrm{scriptSig}]))(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{totalOutputValue}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(\mathrm{fold}^{\lfloor+\rfloor_{64}}((\lambda l.\, l[\mathrm{value}])^{+}(e[\mathrm{tx}][\mathrm{outputs}])))) \\
[\![\mathsf{outputValue}]\!]^{\mathrm{BC}}(i) &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, l[\mathrm{value}])(e[\mathrm{tx}][\mathrm{outputs}]\lceil i \rceil)) \\
[\![\mathsf{outputScriptHash}]\!]^{\mathrm{BC}}(i) &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, \mathrm{SHA256}(l[\mathrm{scriptPubKey}]))(e[\mathrm{tx}][\mathrm{outputs}]\lceil i \rceil)) \\
[\![\mathsf{tapleafVersion}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{tapEnv}][\mathrm{leafVersion}]) \\
[\![\mathsf{tapbranch}]\!]^{\mathrm{BC}}(i) &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{tapEnv}][\mathrm{branch}]\lceil i \rceil) \\
[\![\mathsf{internalKey}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{tapEnv}][\mathrm{internalKey}]) \\
[\![\mathsf{scriptCMR}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{scriptCMR}])
\end{aligned}
$$

The sums computed for $[\![\mathsf{totalInputValue}]\!]^{\mathrm{BC}}\langle\rangle$ and $[\![\mathsf{totalOutputValue}]\!]^{\mathrm{BC}}\langle\rangle$ never "overflow" their 64-bit values due to our assumptions about Bitcoin's money supply.

4.4.1.2 Merkle Roots

We extend the definition of the commitment Merkle root and identity Merkle root to support the new expressions by using the initial value of with tags of new unique byte strings.

$$
\begin{aligned}
\#^{\alpha}(\mathsf{version}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[76657273696f6e]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{lockTime}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[6c6f636b54696d65]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{totalInputValue}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[746f74616c496e70757456616c7565]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{currentIndex}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[63757272656e74496e646578]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{inputPrevOutpoint}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[696e707574507265764f7574706f696e74]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{inputValue}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[696e70757456616c7565]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{inputSequence}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[696e707574536571756e6365]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{inputAnnexHash}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[696e707574416e6e657848617368]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{inputScriptSigHash}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[696e70757453637269707453696748617368]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{totalOutputValue}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[746f74616c4f757470757456616c7565]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{outputValue}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[6f757470757456616c7565]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{outputScriptHash}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[6f75747075745363726970744861736 8]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{tapleafVersion}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[7461706c65616656657273696f6e]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{tapbranch}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[7461706272616e6368]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{internalKey}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[696e7465726e616c4b6579]}}_{\mathrm{IV}} \\
\#^{\alpha}(\mathsf{scriptCMR}) &:= \mathrm{SHA256}^{\mathrm{BCprefix}\cdot\texttt{[736372697074434d52]}}_{\mathrm{IV}}
\end{aligned}
$$

where

$$\mathrm{BCprefix} := \texttt{[53696d706c69636974792d44726166741f5072696d69746976651f426974636f696e1f]}$$

4.5 Simplicity Programs

Ultimately, we only are interested in the side-effects of Simplicity expressions; we only care that a particular expression does not fail when executed within the context of a given transaction. We do not care about the output value of a Simplicity expression, nor do we provide explicit inputs to Simplicity expressions, which are handled by witness expressions instead.

To this end, we define a *Simplicity program* to be a Simplicity expression of type $\mathbb{1} \vdash \mathbb{1}$. A core Simplicity expression of this type is useless. However, a Simplicity program with witnesses, assertions, and Bitcoin, $t : \mathbb{1} \vdash \mathbb{1}$, has semantics of $[\![t]\!]^{\mathrm{BC}}\langle\rangle : \mathrm{BCEnv} \to \mathbb{2}$, which is the type of predicates over BCEnv. This is exactly what we want to use a Blockchain program for: to decide if a given set of witness data authorizes the redemption of funds for a specific input of a specific transaction. A particular input authorizes the transaction in the context $e : \mathrm{BCEnv}$ only when $[\![t]\!]^{\mathrm{BC}}\langle\rangle(e) = \mathtt{1}_{\mathbb{2}}$, and all inputs must authorize the transaction for the transaction to be valid.

Let us look at a basic example of a Simplicity program that requires a single Schnorr signature.

4.5.1 Example: checkSigHashAll

Using Simplicity with witnesses, assertions and Bitcoin, we are able to build an expression that use Schnorr signatures to authorize spending of funds. Using the assertion extension we are able to define a variant of bip-0340-check called bip-0340-verify:

$$\mathsf{bip\text{-}0340\text{-}verify} := \mathsf{assert}\ \mathsf{bip\text{-}0340\text{-}check} : (\mathrm{PubKey} \times \mathrm{Msg}) \times \mathrm{Sig} \vdash \mathbb{1}$$

such that

$$[\![\mathsf{bip\text{-}0340\text{-}verify}]\!]^{\mathrm{S}}\langle\langle p, m\rangle, s\rangle = \eta^{\mathrm{S}}\langle\rangle \;\Leftrightarrow\; [\![\mathsf{bip\text{-}0340\text{-}check}]\!]\langle\langle p, m\rangle, s\rangle = \mathtt{1}_{\mathbb{2}}.$$

Next, we use the Bitcoin transaction extension to build a hashAll expression that computes a SHA-256 hash that commits to all of the current transaction data from the environment and which input is being signed for.

$$\begin{aligned}
&\mathsf{hashAll} : \mathbb{1} \vdash \mathbb{2}^{256}\\
&\mathsf{hashAll} := (\mathsf{scribe}(\mathrm{SHA256}_{\mathrm{IV}}^{\mathtt{[53696d706c69636974792d44726166741f53696748617368]}})\\
&\qquad \vartriangle\ (\mathsf{inputsHash} \vartriangle \mathsf{outputsHash})\\
&\qquad ;\ \mathsf{sha256\text{-}block})\\
&\qquad \vartriangle\ (((\mathsf{currentValue} \vartriangle (\mathsf{currentIndex} \vartriangle \mathsf{lockTime}))\\
&\qquad \vartriangle\ ((\mathsf{version} \vartriangle \mathsf{scribe}(\lfloor 2^{31} \rfloor_{32})) \vartriangle \mathsf{scribe}(\lfloor 0 \rfloor_{64}))) \vartriangle \mathsf{scribe}(\lfloor 1184 \rfloor_{256}))\\
&\qquad ;\ \mathsf{sha256\text{-}block}
\end{aligned}$$

The hashAll expression derives a total of 672 bits of data from the environment. The initial SHA256 adds one 512-bit block constant prefix to this, which totals 1184 bits of data. While it is not strictly necessary, we choose to explicitly append the SHA-256 padding for this length of data.

Next we pair the commitment Merkle root of hashAll with its output. This commitment Merkle root plays the role of the signature hash flag that is covered by the signature in Bitcoin script. It is a specification of which parts of the transaction data is being signed, and exactly how those parts are combined to form the digest being signed.

$$\begin{aligned}
&\mathsf{sigHashAll} : \mathbb{1} \vdash \mathbb{2}^{256}\\
&\mathsf{sigHashAll} := \mathsf{scribe}(\mathrm{SHA256}_{\mathrm{IV}}^{\mathtt{[53696d706c69636974792d44726166741f5369676e6174757265]}})\\
&\qquad \vartriangle\ (\mathsf{scribe}(\#^{c}(\mathsf{hashAll})) \vartriangle \mathsf{hashAll})\\
&\qquad ;\ \mathsf{sha256\text{-}block}) \vartriangle \mathsf{scribe}(\lfloor 2^{511} + 1024 \rfloor_{512})\\
&\qquad ;\ \mathsf{sha256\text{-}block}
\end{aligned}$$

The pair of the commitment Merkle root and the digest itself is 512 bits. The initial SHA256 adds one 512-bit block constant prefix to this, which totals 1024 bits of data. While it is not strictly necessary, we choose to explicitly append the SHA-256 padding for this length of data.

Finally, given a Schnorr public key $p : \mathrm{PubKey}$ and a Schnorr signature $s : \mathrm{Sig}$ we can create a Simplicity program that checks the signature against the public key and above message digest.

$$\begin{aligned}&\mathsf{checkSigHashAll}\langle p, s\rangle : \mathbb{1} \vdash \mathbb{1}\\ &\mathsf{checkSigHashAll}\langle p, s\rangle \;:=\; (\mathsf{scribe}_{\mathrm{PubKey}}(p) \triangle \mathsf{sigHashAll}) \triangle \mathsf{witness}_{\mathrm{Sig}}(s)\\ &\qquad\qquad\qquad\quad ;\; \mathsf{bip\text{-}0340\text{-}verify}\end{aligned}$$

The witness combinator ensures that the program's commitment Merkle root $\#^{c}(\mathsf{checkSigHashAll}\langle p,\, s\rangle)$ is independent of the value of the signature s. This allows us to commit to this program without commiting to the signature, and only providing the signature at redemption time. As with normal Bitcoin transactions, the signature is only valid in the context, $e : \mathrm{BCEnv}$, of a particular input on a particular transaction during redemption because our program only executes successfully, i.e. $[\![\mathsf{checkSigHashAll}\langle p, s\rangle]\!]^{\mathrm{BC}}\langle\rangle(e) = \eta^{\mathrm{S}}\langle\rangle$, when provided a witness that is a valid signature on the transaction data and index number.

4.6 Schnorr Signature Aggregation

4.7 Malleability

4.7.1 Transaction Weight

Chapter 5
Jets

Reminder: build jets for parsing lockTimes.

Chapter 6
Delegation

Our last Simplicity extension is the disconnect combinator. This extension allows for delegation but using it loses some nice properties of Simplicity. The language that uses this extension is called *Simplicity with delegation*. The language that uses this and all other extensions is called *full Simplicity with delegation*.

$$\frac{s : 2^{256} \times A \vdash B \times C \qquad t : C \vdash D}{\mathsf{disconnect}_{A,B,C,D}\, s\, t : A \vdash B \times D}$$

Semantically, the disconnect combinator behaves similar to the composition combinator, but where the commitment Merkle root of the expression t is passed as an argument to the expression s. We extend our formal semantics to the disconnect combinator by defining it in terms of core Simplicity as follows.

$$[\![\mathsf{disconnect}_{A,B,C,D}\, s\, t]\!]^{\mathcal{M}} := [\![\mathsf{scribe}_{A,2^{256}}(\#^c(t)) \vartriangle \mathsf{iden}; s; \mathsf{take}\,\mathsf{iden} \vartriangle \mathsf{drop}\, t]\!]^{\mathcal{M}}$$

We can simplify the semantics as follows.

$$[\![\mathsf{disconnect}_{A,B,C,D}\, s\, t]\!]^{\mathcal{M}}(a) = ([\![\mathsf{take}\,\mathsf{iden} \vartriangle \mathsf{drop}\, t]\!]^{\mathcal{M}} \leftarrowtail [\![s]\!]^{\mathcal{M}})\langle \#^c(t), a\rangle$$

Like a witness expression, the real significance comes from the form of its commitment Merkle root. We extend the definition of the commitment Merkle root as follows.

$$\#^c(\mathsf{disconnect}_{A,B,C,D}\, s\, t) \;:=\; \mathrm{SHA256}_{\mathrm{Block}}\langle \mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^c_{\mathsf{disconnect}}}, \langle \lfloor 0 \rfloor_{256}, \#^c(s)\rangle\rangle$$

where the $\mathrm{tag}^c_{\mathsf{disconnect}}$ value is a unique value.

$$\mathrm{tag}^c_{\mathsf{disconnect}} \;:=\; \texttt{[53696d706c69636974792d44726166741f436f6d6d69746d656e741f646973636f6e6e656374]}_{2^8}$$

The commitment Merkle root only commits to the first argument, s, of a disconnect $s\,t$ expression. During redemption the second argument, t, can be freely set to any Simplicity expression. This differs from the identity root (Section 7.2.3) which will include a commitment to both arguments. In order to place restrictions on what t is allowed, the commitment Merkle root of t is passed to s as an input. This way s is allowed to dynamically decide if t is an acceptable expression to be used here.

The primary purpose of disconnect is for delegation. In this scenario, s, validates that the commitment Merkle root $\#^c(t)$ is signed by a fixed public key. This lets a user postpone defining t until redemption time, while still maintaining full control, because any redemption requires their signature on $\#^c(t)$. For example, a user can require that $[\![t]\!]^{\mathcal{M}}$ returns a public key. After commitment, but before redemption, the user can delegate authorization to redeem these funds to a third party by signing the that party's key in this fashion.

The disconnect combinator comes with some significant caveats. Because the whole program is not committed to at commitment time, it is no longer possible to statically analyze the maximum resource costs for redemption before commitment. During redemption $[\![t]\!]^{\mathcal{M}}$ could, a priori, perform arbitrary amounts computation. Indeed we can use disconnect to commit to what are effectively unbounded loops in Simplicity (see Section 6.2). Contrast this with the witness expression, where type inference limits the size of the witness data, so bounds are still possible to compute.

Of course, depending on the specifics of the policy enforced by s, it may be possible to bound the maximum resource costs for redemption in specific cases. However, that depends on the details of s and there is no practical, universal algorithm that will work for any Simplicity expression. Using disconnect risks creating program that ends up impractical to redeem due to costs. This danger is why disconnect is not part of full Simplicity and it is instead considered an extension to be used with caution.

However, it is also important to note that static analysis can be performed at redemption time. At that point time the t expression has been provided and usual static analysis can proceed. Static analysis can still be part of the consensus protocol, even when the disconnect expression is used.

6.1 Implementing disconnect on the Bit Machine

The semantics of the disconnect combinator can tell us how to implement it on the Bit Machine. We simply define the translation of the disconnect combinator to be the translation of its semantics.

$$\begin{aligned}
\langle\!\langle \mathsf{disconnect}_{A,B,C,D}\, s\, t \rangle\!\rangle &:= \langle\!\langle \mathsf{scribe}_{A,2^{256}}(\#^c(t)) \vartriangle \mathsf{iden}; (s; \mathsf{take}\,\mathsf{iden} \vartriangle \mathsf{drop}\, t) \rangle\!\rangle \\
\langle\!\langle \mathsf{disconnect}_{A,B,C,D}\, s\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} &:= \langle\!\langle \mathsf{scribe}_{A,2^{256}}(\#^c(t)) \vartriangle \mathsf{iden}; (s; \mathsf{take}\,\mathsf{iden} \vartriangle \mathsf{drop}\, t) \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \\
\langle\!\langle \mathsf{disconnect}_{A,B,C,D}\, s\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} &:= \langle\!\langle \mathsf{scribe}_{A,2^{256}}(\#^c(t)) \vartriangle \mathsf{iden}; (s; \mathsf{take}\,\mathsf{iden} \vartriangle \mathsf{drop}\, t) \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}}
\end{aligned}$$

We can expand out each of these definitions in turn to see their definition in terms of Bit Machine operations:

$$\begin{aligned}
\langle\!\langle \mathsf{disconnect}_{A,B,C,D}\, s\, t \rangle\!\rangle &= \mathrm{newFrame}(256 + \mathrm{bitSize}(A)) \\
&\;;\; \mathrm{write}_{2^{256}}(\#^c(t)); \mathrm{copy}(\mathrm{bitSize}(A)) \\
&\;;\; \mathrm{moveFrame} \\
&\;;\; \mathrm{newFrame}(\mathrm{bitSize}(B) + \mathrm{bitSize}(C)) \\
&\;;\; \langle\!\langle s \rangle\!\rangle \\
&\;;\; \mathrm{moveFrame} \\
&\;;\; \mathrm{copy}(\mathrm{bitSize}(B)); \mathrm{bitSize}(B) \star \langle\!\langle t \rangle\!\rangle \\
&\;;\; \mathrm{dropFrame} \\
&\;;\; \mathrm{dropFrame} \\
\langle\!\langle \mathsf{disconnect}_{A,B,C,D}\, s\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} &= \mathrm{newFrame}(256 + \mathrm{bitSize}(A)) \\
&\;;\; \mathrm{write}_{2^{256}}(\#^c(t)); \mathrm{copy}(\mathrm{bitSize}(A)) \\
&\;;\; \mathrm{moveFrame} \\
&\;;\; \mathrm{newFrame}(\mathrm{bitSize}(B) + \mathrm{bitSize}(C)) \\
&\;;\; \langle\!\langle s \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
&\;;\; \mathrm{moveFrame} \\
&\;;\; \mathrm{copy}(\mathrm{bitSize}(B)); \mathrm{fwd}(\mathrm{bitSize}(B)); \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
\langle\!\langle \mathsf{disconnect}_{A,B,C,D}\, s\, t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} &= \mathrm{newFrame}(256 + \mathrm{bitSize}(A)) \\
&\;;\; \mathrm{write}_{2^{256}}(\#^c(t)); \mathrm{copy}(\mathrm{bitSize}(A)) \\
&\;;\; \mathrm{dropFrame} \\
&\;;\; \mathrm{moveFrame} \\
&\;;\; \mathrm{newFrame}(\mathrm{bitSize}(B) + \mathrm{bitSize}(C)) \\
&\;;\; \langle\!\langle s \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \\
&\;;\; \mathrm{moveFrame} \\
&\;;\; \mathrm{copy}(\mathrm{bitSize}(B)); \mathrm{fwd}(\mathrm{bitSize}(B)); \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}}
\end{aligned}$$

where

$$\begin{aligned}
\mathrm{write}_{2^1}(0_2) &:= \mathrm{write}(0); \mathrm{skip}(0); \mathrm{nop} \\
\mathrm{write}_{2^1}(1_2) &:= \mathrm{write}(1); \mathrm{skip}(0); \mathrm{nop} \\
\mathrm{write}_{2^{2n}}(\langle a, b \rangle) &:= \mathrm{write}_{2^n}(a); \mathrm{write}_{2^n}(b)
\end{aligned}$$

Of course, in an optimized implementation of disconnect we would discard the skip(0) and nop operations from the definition of $\mathrm{write}_{2^{256}}$. TODO: before doing Time Resource static analysis we may need to formally define this optimized implemenation of disconnect.

Because the Bit Machine instructions are directly derived from the semantics of disconnect the correctness theorems for the Bit Machine still hold.

Theorem 6.1. *Given a well-typed Simplicity program with delegation $t\colon A \vdash B$ and an input $a\colon A$, then*

$$[\Theta \rhd r_0 \wedge \ulcorner a \urcorner \cdot r_0' | w_0 \wedge \texttt{[?]}^{\,\mathrm{bitSize}(B)+m} \lhd \Xi] \nrightarrow \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [\Theta \rhd r_0 \wedge \ulcorner a \urcorner \cdot r_0' | w_0 \cdot \ulcorner [\![t]\!](a) \urcorner \wedge \texttt{[?]}^{\,m} \lhd \Xi]$$

for any cell arrays r_0, r_0', w_0, any stacks Θ, Ξ, and any natural number m.

In particular, for a well-typed Simplicity program with delegation $t\colon A \vdash B$, we have

$$[\wedge \ulcorner a \urcorner | \wedge \texttt{[?]}^{\,\mathrm{bitSize}(B)}] \nrightarrow \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [\wedge \ulcorner a \urcorner | \ulcorner [\![t]\!](a) \urcorner \wedge]$$

Theorem 6.2. *Given a well-typed Simplicity program with delegation $t\colon A \vdash B$ and an input $a\colon A$, then*

$$[\Theta \rhd r_0 \wedge \ulcorner a \urcorner \cdot r_0' | w_0 \wedge \texttt{[?]}^{\,\mathrm{bitSize}(B)+m} \lhd \Xi] \nrightarrow \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \twoheadrightarrow [\Theta \rhd r_0 \wedge \ulcorner a \urcorner \cdot r_0' | w_0 \cdot \ulcorner [\![t]\!](a) \urcorner \wedge \texttt{[?]}^{\,m} \lhd \Xi]$$

and

$$[\Theta \rhd r_1 \rhd r_0 \wedge \ulcorner a \urcorner \cdot r_0' | w_0 \wedge \texttt{[?]}^{\,\mathrm{bitSize}(B)+m} \lhd \Xi] \nrightarrow \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \twoheadrightarrow [\Theta \rhd r_1 | w_0 \cdot \ulcorner [\![t]\!](a) \urcorner \wedge \texttt{[?]}^{\,m} \lhd \Xi]$$

for any cell arrays r_0, r_0', w_0, any frame r_1, any stacks Θ, Ξ, and any natural number m.

In particular, for a well-typed Simplicity program with delegation $t\colon A \vdash B$, we have

$$[\wedge \ulcorner a \urcorner | \wedge \texttt{[?]}^{\,\mathrm{bitSize}(B)}] \nrightarrow \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \twoheadrightarrow [\wedge \ulcorner a \urcorner | \ulcorner [\![t]\!](a) \urcorner \wedge]$$

6.1.1 Static Analysis of disconnect

6.1.1.1 Space Resources

Because the Bit Machine implementation of the disconnect combinator is derived from its semantics in core Simplicity, we can derive the static analysis of the space resources used by the Bit Machine implementation of the disconnect combintor from its definining Simplicity expression.

$$\begin{aligned}
\mathrm{extraCellsBound}(\mathsf{disconnect}_{A,B,C,D}\, s\, t) &:= \mathrm{extraCellsBound}\\
&\qquad (\mathsf{scribe}_{A,2^{256}}(\#^c(t)) \vartriangle \mathsf{iden}; (s; \mathsf{take}\,\mathsf{iden} \vartriangle \mathsf{drop}\, t))\\
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}(\mathsf{disconnect}_{A,B,C,D}\, s\, t) &:= \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}\\
&\qquad (\mathsf{scribe}_{A,2^{256}}(\#^c(t)) \vartriangle \mathsf{iden}; (s; \mathsf{take}\,\mathsf{iden} \vartriangle \mathsf{drop}\, t))\\
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(\mathsf{disconnect}_{A,B,C,D}\, s\, t) &:= \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}\\
&\qquad (\mathsf{scribe}_{A,2^{256}}(\#^c(t)) \vartriangle \mathsf{iden}; (s; \mathsf{take}\,\mathsf{iden} \vartriangle \mathsf{drop}\, t))
\end{aligned}$$

We can expand out each of these definitions in turn:

$$\begin{aligned}
\mathrm{extraCellsBound}(\mathsf{disconnect}_{A,B,C,D}\, s\, t) &= 256 + \mathrm{bitSize}(A)\\
&+ \mathrm{bitSize}(B) + \mathrm{bitSize}(C)\\
&+ \max(\mathrm{extraCellsBound}(s), \mathrm{extraCellsBound}(t))
\end{aligned}$$

$$\begin{aligned}
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}(\mathsf{disconnect}_{A,B,C,D}\, s\, t)(r) &= \max(r_a, \max(\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}(s)(r_a) + r_a,\\
&\qquad \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}(t)(r_b)) + r_b - r)\\
&\text{where } r_a := 256 + \mathrm{bitSize}(A)\\
&\text{and } r_b := \mathrm{bitSize}(B) + \mathrm{bitSize}(C)
\end{aligned}$$

$$\begin{aligned}\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(\mathsf{disconnect}_{A,B,C,D}\, s\, t) \quad &= \quad \langle \max(n_t, r_b + \max(n_s, m_t, r_a + m_s)), r_a\rangle \\ &\text{where} \quad \langle n_s, m_s\rangle := \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(s) \\ &\text{and} \quad \langle n_t, m_t\rangle := \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t) \\ &\text{and} \quad r_a := 256 + \mathrm{bitSize}(A) \\ &\text{and} \quad r_b := \mathrm{bitSize}(B) + \mathrm{bitSize}(C)\end{aligned}$$

The correctness theorems about these analyses still holds:

Lemma 6.3. *For any Simplicity expression with delegation $t\colon A \vdash B$, such that*

$$[\Theta \rhd r_0 | w_0 \lhd \Xi] \rightarrowtail \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [\Theta' \rhd r'_0 | w'_0 \lhd \Xi']$$

we have that

1. $\mathrm{cellCount}([\Theta \rhd r_0 | w_0 \lhd \Xi]) = \mathrm{cellCount}([\Theta' \rhd r'_0 | w'_0 \lhd \Xi'])$
2. $\mathrm{cellsReq}([\Theta \rhd r_0 | w_0 \lhd \Xi] \rightarrowtail \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [\Theta' \rhd r'_0 | w'_0 \lhd \Xi']) \leq \mathrm{cellCount}([\Theta \rhd r_0 | w_0 \lhd \Xi]) + \mathrm{extraCellsBound}(t).$

In particular for $a\colon A$ and

$$[{}_\wedge\ulcorner a \urcorner_\wedge [\mathtt{?}]^{\,\mathrm{bitSize}(B)}] \rightarrowtail \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [{}_\wedge\ulcorner a \urcorner |\ulcorner [\![t]\!](a) \urcorner_\wedge]$$

we have that

$\mathrm{cellsReq}([{}_\wedge\ulcorner a \urcorner_\wedge [\mathtt{?}]^{\,\mathrm{bitSize}(B)}] \rightarrowtail \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [{}_\wedge\ulcorner a \urcorner |\ulcorner [\![t]\!](a) \urcorner_\wedge]) \leq \mathrm{bitSize}(A) + \mathrm{bitSize}(B) + \mathrm{extraCellsBound}(t).$

Lemma 6.4. *For any Simplicity expression with delegation $t\colon A \vdash B$, such that*

$$[\Theta_{\mathrm{on}} \rhd r_{\mathrm{on},0} | w_{\mathrm{on},0} \lhd \Xi_{\mathrm{on}}] \rightarrowtail \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \twoheadrightarrow [\Theta'_{\mathrm{on}} \rhd r'_{\mathrm{on},0} | w'_{\mathrm{on},0} \lhd \Xi'_{\mathrm{on}}]$$

and

$$[\Theta_{\mathrm{off}} \rhd r_{\mathrm{off},0} | w_{\mathrm{off},0} \lhd \Xi_{\mathrm{off}}] \rightarrowtail \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \twoheadrightarrow [\Theta'_{\mathrm{off}} \rhd r'_{\mathrm{off},0} | w'_{\mathrm{off},0} \lhd \Xi'_{\mathrm{off}}]$$

we have that

1. $\mathrm{cellCount}([\Theta_{\mathrm{on}} \rhd r_{\mathrm{on},0} | w_{\mathrm{on},0} \lhd \Xi_{\mathrm{on}}]) = \mathrm{cellCount}(r_{\mathrm{on},0}) + \mathrm{cellCount}([\Theta'_{\mathrm{on}} \rhd r'_{\mathrm{on},0} | w'_{\mathrm{on},0} \lhd \Xi'_{\mathrm{on}}])$ *and*
   $\mathrm{cellCount}([\Theta_{\mathrm{off}} \rhd r_{\mathrm{off},0} | w_{\mathrm{off},0} \lhd \Xi_{\mathrm{off}}]) = \mathrm{cellCount}([\Theta'_{\mathrm{off}} \rhd r'_{\mathrm{off},0} | w'_{\mathrm{off},0} \lhd \Xi'_{\mathrm{off}}])$
2. $\mathrm{cellsReq}([\Theta_{\mathrm{on}} \rhd r_{\mathrm{on},0} | w_{\mathrm{on},0} \lhd \Xi_{\mathrm{on}}] \rightarrowtail \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}} \twoheadrightarrow [\Theta'_{\mathrm{on}} \rhd r'_{\mathrm{on},0} | w'_{\mathrm{on},0} \lhd \Xi'_{\mathrm{on}}]) \leq \mathrm{cellCount}([\Theta_{\mathrm{on}} \rhd r_{\mathrm{on},0} | w_{\mathrm{on},0} \lhd \Xi_{\mathrm{on}}]) + \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}(t)(\mathrm{cellCount}(r_{\mathrm{on},0}))$ *and*
   $\mathrm{cellsReq}([\Theta_{\mathrm{off}} \rhd r_{\mathrm{off},0} | w_{\mathrm{off},0} \lhd \Xi_{\mathrm{off}}] \rightarrowtail \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \twoheadrightarrow [\Theta'_{\mathrm{off}} \rhd r'_{\mathrm{off},0} | w'_{\mathrm{off},0} \lhd \Xi'_{\mathrm{off}}]) \leq \mathrm{cellCount}([\Theta_{\mathrm{off}} \rhd r_{\mathrm{off},0} | w_{\mathrm{off},0} \lhd \Xi_{\mathrm{off}}]) + \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}(t)(0).$

In particular for $a\colon A$ and

$$[{}_\wedge\ulcorner a \urcorner_\wedge [\mathtt{?}]^{\,\mathrm{bitSize}(B)}] \rightarrowtail \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \twoheadrightarrow [{}_\wedge\ulcorner a \urcorner |\ulcorner [\![t]\!](a) \urcorner_\wedge]$$

we have that

$\mathrm{cellsReq}([{}_\wedge\ulcorner a \urcorner_\wedge [\mathtt{?}]^{\,\mathrm{bitSize}(B)}] \rightarrowtail \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \twoheadrightarrow [{}_\wedge\ulcorner a \urcorner |\ulcorner [\![t]\!](a) \urcorner_\wedge]) \leq \mathrm{bitSize}(A) + \mathrm{bitSize}(B) + \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}(t)(0).$

Lemma 6.5. *For any Simplicity expression with delegation $t\colon A \vdash B$,*

$$\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}(t) = \mathrm{interp}^{\mathrm{TCO}}(\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t)).$$

Corollary 6.6. *For any Simplicity expression with delegation $t\colon A \vdash B$ and $a\colon A$ such that*

$$[{}_\wedge\ulcorner a \urcorner_\wedge [\mathtt{?}]^{\,\mathrm{bitSize}(B)}] \rightarrowtail \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \twoheadrightarrow [{}_\wedge\ulcorner a \urcorner |\ulcorner [\![t]\!](a) \urcorner_\wedge]$$

we have that

$\mathrm{cellsReq}([{}_\wedge\ulcorner a \urcorner_\wedge [\mathtt{?}]^{\,\mathrm{bitSize}(B)}] \rightarrowtail \langle\!\langle t \rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{off}} \twoheadrightarrow [{}_\wedge\ulcorner a \urcorner |\ulcorner [\![t]\!](a) \urcorner_\wedge]) \leq \mathrm{bitSize}(A) + \mathrm{bitSize}(B) + \max(n, m)$

where $\langle n, m\rangle := \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t)$.

6.2 Unbounded Loops

While the primary purpose of disconnect is for delegation, the construct can be used to create what is effectively an unbounded loop in Simplicity. Given a Simplicity expression, $t : A \vdash A + B$, we can build a commitment Merkle root to an expression that will repeatedly recursively evaluate $[\![t]\!]\colon A \to A + B$ on an input A until a B value is returned.

Consider a Simplicity expression $t\colon A \vdash A + B$, and a continuation $k\colon A \times 2^{256} \vdash B$. We define $\mathsf{loopBody}\, t\, k$:

$$\begin{aligned}&\mathsf{loopBody}\, t\, k : A \times 2^{256} \vdash B\\ &\mathsf{loopBody}\, t\, k \;:=\; \mathsf{take}\, t \vartriangle \mathsf{IH}\\ &\qquad\qquad ;\; \mathsf{case}\ (\mathsf{disconnect}\ (\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{IH})\, k; \mathsf{IH})\ \mathsf{OH}\end{aligned}$$

Let us consider the semantics $[\![\mathsf{loopBody}\, t\, k]\!]^{\mathcal{M}}$. Given inputs $a_0 : A$ and $h : 2^{256}$, then in the first clause in the definition of loopBody we have

$$[\![\mathsf{take}\, t \vartriangle \mathsf{IH}]\!]^{\mathcal{M}}\langle a_0, h\rangle = \phi^{\mathcal{M}}\langle [\![t]\!]^{\mathcal{M}}(a_0), \eta^{\mathcal{M}}(h)\rangle : \mathcal{M}((A + B) \times 2^{256}).$$

It evaluates $[\![t]\!]^{\mathcal{M}}(a_0)$ and pairs that with h. For those contexts in which $[\![t]\!]^{\mathcal{M}}(a_0)$ doesn't fail, it results in either an $\sigma^{\mathbf{L}}(a_1) : A + B$ for some $a_1 : A$, or $\sigma^{\mathbf{R}}(b) : A + B$ from some $b : B$. Let us consider the easy case, $\sigma^{\mathbf{R}}(b)$, first. The remainder of loopBody continues as follows.

$$\begin{aligned}&[\![\mathsf{case}\ (\mathsf{disconnect}\ (\mathsf{assert}\ (\mathsf{OIH} \vartriangle \mathsf{IH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{OH})\, k; \mathsf{IH})\ \mathsf{OH}]\!]^{\mathcal{M}}\langle \sigma^{\mathbf{R}}(b), h\rangle\\ =\;& [\![\mathsf{OH}]\!]^{\mathcal{M}}\langle b, h\rangle\\ =\;& \eta^{\mathcal{M}}(b)\end{aligned}$$

Whenever $[\![t]\!]^{\mathcal{M}}(a_0)$, "returns" a $\sigma^{\mathbf{R}}(b)$ value, that b value is the "result" of $[\![\mathsf{loopBody}\, t\, k]\!]$, and k is ignored.

Now let us consider the case when $[\![t]\!]^{\mathcal{M}}(a_0)$ results in an $\sigma^{\mathbf{L}}(a_1)$ value. In this case the remainder of loopBody continues as follows.

$$\begin{aligned}&[\![\mathsf{case}\ (\mathsf{disconnect}\ (\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{IH})\, k; \mathsf{IH})\ \mathsf{OH}]\!]^{\mathcal{M}}\langle \sigma^{\mathbf{L}}(a_1), h\rangle\\ =\;& [\![\mathsf{disconnect}\ (\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{IH})\, k; \mathsf{IH}]\!]^{\mathcal{M}}\langle a_1, h\rangle\\ =\;& ([\![\mathsf{IH}]\!]^{\mathcal{M}} \leftharpoonup [\![\mathsf{disconnect}\ (\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{IH})\, k]\!]^{\mathcal{M}})\langle a_1, h\rangle\\ =\;& ([\![\mathsf{IH}]\!]^{\mathcal{M}} \leftharpoonup [\![\mathsf{take}\, \mathsf{iden} \vartriangle \mathsf{drop}\, k]\!]^{\mathcal{M}} \leftharpoonup [\![\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{IH}]\!]^{\mathcal{M}})\langle \#^c(k), \langle a_1, h\rangle\rangle\\ =\;& ([\![\mathsf{take}\, \mathsf{iden} \vartriangle \mathsf{drop}\, k; \mathsf{IH}]\!]^{\mathcal{M}} \leftharpoonup [\![\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{IH}]\!]^{\mathcal{M}})\langle \#^c(k), \langle a_1, h\rangle\rangle\\ =\;& ([\![\mathsf{drop}\, k]\!]^{\mathcal{M}} \leftharpoonup [\![\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{IH}]\!]^{\mathcal{M}})\langle \#^c(k), \langle a_1, h\rangle\rangle\end{aligned}$$

For the first part we have

$$\begin{aligned}&[\![\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{IH}]\!]^{\mathcal{M}}\langle \#^c(k), \langle a_1, h\rangle\rangle\\ =\;& \iota^{\mathcal{M}}_{\mathrm{S}}([\![\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{IH}]\!]^{\mathrm{S}}\langle \#^c(k), \langle a_1, h\rangle\rangle)\\ =\;& \iota^{\mathcal{M}}_{\mathrm{S}}(\phi^{\mathrm{S}}\langle [\![\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}})]\!]^{\mathrm{S}}\langle \#^c(k), \langle a_1, h\rangle\rangle, \eta^S\langle a_1, h\rangle\rangle)\\ =\;& \iota^{\mathcal{M}}_{\mathrm{S}}(\phi^{\mathrm{S}}\langle [\![\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}]\!]\langle \#^c(k), \langle a_1, h\rangle\rangle, \eta^S\langle a_1, h\rangle\rangle)\\ =\;& \iota^{\mathcal{M}}_{\mathrm{S}}(\phi^{\mathrm{S}}\langle [\![\mathsf{eq}_{2^{256}}]\!]\langle h, \#^c(k)\rangle, \eta^S\langle a_1, h\rangle\rangle)\end{aligned}$$

We know that $[\![\mathsf{eq}_{2^{256}}]\!]\langle h, \#^c(k)\rangle = \mathsf{1}_2 = \eta^S\langle\rangle$ if and only if $h = \#^c(k)$ and that $[\![\mathsf{eq}_{2^{256}}]\!]\langle h, \#^c(k)\rangle = \mathsf{0}_2 = \emptyset^S$ if and only if $h \neq \#^c(k)$. When $h \neq \#^c(k)$, then

$$[\![\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{IH}]\!]^{\mathcal{M}}\langle\langle a_1, h\rangle, \#^c(k)\rangle = \emptyset^{\mathcal{M}}$$

and the whole loopBody expression fails with a $\emptyset^{\mathcal{M}}_B$ result. However, when $h = \#^c(k)$ we have that

$$[\![\mathsf{assert}\ (\mathsf{IIH} \vartriangle \mathsf{OH}\,;\mathsf{eq}_{2^{256}}) \vartriangle \mathsf{IH}]\!]^{\mathcal{M}}\langle\langle a_1, h\rangle, \#^c(k)\rangle = \eta^{\mathcal{M}}\langle\langle\rangle, \langle a_1, h\rangle\rangle$$

and we can continue with

$$[\![\mathsf{drop}\,k]\!]^{\mathcal{M}}\langle\langle\rangle,\langle a_1,h\rangle\rangle \;=\; [\![k]\!]^{\mathcal{M}}\langle a_1,h\rangle$$

and the result is that

$$[\![\mathsf{case}\,(\mathsf{disconnect}\,(\mathsf{assert}\,(\mathsf{IIH}\vartriangle\mathsf{OH}\,;\mathsf{eq}_{2^{256}})\vartriangle\mathsf{IH})\,k;\mathsf{IH})\,\mathsf{OH}]\!]^{\mathcal{M}}\langle\sigma^{\mathbf{L}}(a_1),h\rangle = [\![k]\!]^{\mathcal{M}}\langle a_1,h\rangle.$$

Recapping, when $[\![t]\!]^{\mathcal{M}}(a_0)$ results in an $\sigma^{\mathbf{L}}(a_1)$, then $[\![\mathsf{loopBody}\,t\,k]\!]^{\mathcal{M}}\langle a_0,h\rangle$ evaluates the continuation $[\![k]\!]^{\mathcal{M}}\langle a_1,h\rangle$ under the condition that the h value matches the commitment Merkle root of k, i.e. $h=\#^c(k)$, and fails when $h\neq\#^c(k)$.

Now let us consider loopBody's commitment Merkle root, $\#^c(\mathsf{loopBody}\,t\,k)$. Because of the use of disconnect, this commitment Merkle root is independent of the expression k.

Lemma 6.7. *For all $t:A\vdash A+B$ and $k:A\times 2^{256}\vdash B$, $\#^c(\mathsf{loopBody}\,t\,k)=\mathrm{loopBodyCMR}(t)$ where*

$$\begin{aligned}\mathrm{loopBodyCMR}(t) \;:=\; &\mathrm{SHA256}_{\mathrm{Block}}\big\langle\mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^c_{\mathsf{comp}}},\langle\#^c(\mathsf{take}\,t\vartriangle\mathsf{IH}),\\ &\mathrm{SHA256}_{\mathrm{Block}}\langle\mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^c_{\mathsf{case}}},\langle\mathrm{loopTail},\#^c(\mathsf{OH})\rangle\rangle\rangle\big\rangle\end{aligned}$$

and

$$\begin{aligned}\mathrm{loopTail} \;:=\; &\mathrm{SHA256}_{\mathrm{Block}}\big\langle\mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^c_{\mathsf{comp}}},\big\langle\\ &\mathrm{SHA256}_{\mathrm{Block}}\big\langle\mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^c_{\mathsf{disconnect}}},\langle\lfloor 0\rfloor_{256},\#^c(\mathsf{assert}\,(\mathsf{IIH}\vartriangle\mathsf{OH}\,;\mathsf{eq}_{2^{256}})\vartriangle\mathsf{IH})\rangle\big\rangle,\#^c(\mathsf{IH})\big\rangle\big\rangle.\end{aligned}$$

While the commitment Merkle root of $\mathsf{loopBody}\,t\,k$ does not restrict what the continuation k is, the second component of the input to the expression does, since $h=\#^c(k)$ is required for evaluation to be successful. We can build a new expression that forces the continuation k to be $\mathsf{loopBody}\,t\,k$ itself:

$$\begin{aligned}&\mathsf{loop}\,t\,k:A\vdash B\\ &\mathsf{loop}\,t\,k \;:=\; \mathsf{iden}\vartriangle\mathrm{scribe}(\mathrm{loopBodyCMR}(t));\mathsf{loopBody}\,t\,k\end{aligned}$$

Again, loop's commitment Merkle root is independent of the continuation k:

Lemma 6.8. *For all $t:A\vdash A+B$ and $k:A\times 2^{256}\vdash B$, $\#^c(\mathsf{loop}\,t\,k)=\mathrm{loopCMR}(t)$ where*

$$\mathrm{loopCMR}(t) \;:=\; \mathrm{SHA256}_{\mathrm{Block}}\big\langle\mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^c_{\mathsf{comp}}},\langle\#^c(\mathsf{iden}\vartriangle\mathrm{scribe}(\mathrm{loopBodyCMR}(t))),\mathrm{loopBodyCMR}(t)\rangle\big\rangle.$$

By design, $[\![\mathsf{loop}\;t\,k]\!]^{\mathcal{M}}(a_0)$ passes loopBodyCMR(t) as the h value to $\mathsf{loopBody}\;t\,k$, and the result can only be successful when $\#^c(k)=\mathrm{loopBodyCMR}(t)=\#^c(\mathsf{loopBody}\;t\,k')$ for some further continuation $k':A\times 2^{256}\vdash B$. Furthermore, the same h value, is passed again into $[\![k]\!]^{\mathcal{M}}\langle a_1,h\rangle=[\![\mathsf{loopBody}\;t\,k']\!]^{\mathcal{M}}\langle a_1,h\rangle$, and that result can only be successful when $\#^c(k')=\#^c(\mathsf{loopBody}\;t\,k'')$ for some yet further continuation $k'':A\times 2^{256}\vdash B$.

This process looks endless, making it impossible to redeem any loopCMR(t) commitment. However we can end this chain by using

$$\begin{aligned}&\mathsf{loopEnd}\,t:A\times 2^{256}\vdash B\\ &\mathsf{loopEnd}\,t \;:=\; \mathsf{take}\,t\vartriangle\mathsf{IH}\\ &\qquad\qquad;\;\mathsf{assertr}\;\mathrm{loopTail}\;\mathsf{OH}\end{aligned}$$

The $\mathsf{loopEnd}\,t$ expression replaces the case combinator in $\mathsf{loopBody}\,t$ with an assertion that $[\![t]\!]^{\mathcal{M}}(a_n)$ results in a $\sigma^{\mathbf{R}}(b)$. By construction $\#^c(\mathsf{loopEnd}\,t)=\mathrm{loopBodyCMR}(t)$, so that it can be used as a final continuation to terminate our chain of expressions in $\mathsf{loop}\,t\,k$.

Tying everything together, we have a commitment Merkle root, loopCMR(t), that makes a commitment to a subexpression that can be redeemed by a subexpression of the form

$$\mathsf{loop}\,t\,(\mathsf{loopBody}\,t\,(\mathsf{loopBody}\,t\,(\ldots(\mathsf{loopBody}\,t\,(\mathsf{loopEnd}\,t)))))$$

where loopBody is repeated the number of times necessary for $[\![t]\!]^{\mathcal{M}}(a_n)$ to result in a $\sigma^{\mathbf{R}}(b)$ value in the particular redemption context. Note that while it appears that t is duplicated n times, the subexpression t will be shared in a DAG representation of this expression, avoiding a large of a blow up in program size.

We have repeated the same expression t in during redemption in the above example; however, if the t expression contains witness subexpressions, those witness values could be different for each instance of t shown above. Thus, more generally, the commitment can be redeemed by

$$\mathsf{loop}\, t_0\, (\mathsf{loopBody}\, t_1\, (\mathsf{loopBody}\, t_2\, (\ldots(\mathsf{loopBody}\, t_{n-1}\, (\mathsf{loopEnd}\, t_n)))))$$

when $\#^c(t_i) = \#^c(t)$ for all i. For example, this construction would let you build a Simplicity expression that takes an arbitrarily long stream of SHA-256's 512 bit data blocks and computes the SHA-256 compression function composed over this stream of blocks by putting each data block into a witness expression within the different t_i's.

6.2.1 Adding a loop primitive to Simplicity?

The purpose of the above derivation of the loop construction is to understand the theoretical expressiveness that Simplicity's delegation extension adds to the language. The loop construction proves that the delegation extension brings Simplicity's expressiveness beyond what is possible with full Simplicity without delegation.

While it is, in principle, possible to use the loop construction in Simplicity applications (that support the delegation extension), the loop construction is subject to all the same problems that using delegation entails: One cannot bound the computation costs of redemption at commitment time using general purpose static analysis. One might be able to perform an ad-hoc analysis on a particular program in order to bound the computation costs, but there are programs you can write in Simplicity with delegation where no upper bound on redemption costs exists. As with disconnect, using loop risks creating program that ends up impractical to redeem due to costs. Therefore we strongly recommend against using loop in practice.

If, contrary to our expectations, the loop construction ends up very popular, we could amend the specification of Simplicity to add native $\mathsf{loop}\, t\, k$ and $\mathsf{loopEnd}\, t$ combinators to Simplicity such that their commitment Merkle roots that depends only on t, and with rules that require the continuation k be one of the two loop constructors (with the same t parameter). This would somewhat reduce the overhead of using unbounded loops.

6.3 Universal Signature Hash Modes

In Section 4.5.1 we defined a Simplicity Program for a single signature check with over fixed hashAll transaction digest message. However, in Bitcoin Script, the user can select among a set of multiple different signature hash modes. Moreover, this choice is made at redeption time rather than at commitment time. We could replicate this behaviour by writing a Simplicity program that, given a signature hash flag value defined by a witness node, selects amongst a small variety of transaction digest expressions to use. However by using disconnect, we can provide the user with an unlimited choice of signature hash modes.

Below we define $\mathsf{sigHash}\, t$ where $t : \mathbb{1} \vdash 2^{256}$ is a Simplicity expression that defines some signature hash mode.

$\mathsf{sigHash}\, t : \mathbb{1} \vdash 2^{256}$

$$\begin{aligned}
\mathsf{sigHash}\, t \;&:=\; \mathsf{scribe}(\mathrm{SHA256}_{\mathrm{IV}}^{\texttt{[53696d706c69636974792d44726166741f5369676e6174757265]}}) \\
&\triangle\; \mathsf{disconnect}\, \mathsf{iden}\, t \\
&;\; \mathsf{sha256\text{-}block} \triangle \mathsf{scribe}(\lfloor 2^{511} + 1024 \rfloor_{512}) \\
&;\; \mathsf{sha256\text{-}block}
\end{aligned}$$

This expression can be composed with pubkey key and signature values to form a Simplicity program that supports universal signature hash modes.

$(\mathsf{checkSigHash}\, t)\langle p, s\rangle : \mathbb{1} \vdash \mathbb{1}$

$$\begin{array}{rcl}(\mathsf{checkSigHash}\ t)\langle p, s\rangle & := & (\mathsf{scribe}_{\mathrm{PubKey}}(p) \vartriangle \mathsf{sigHash}\ t) \vartriangle \mathsf{witness}_{\mathrm{Sig}}(s) \\ & ; & \mathsf{bip\text{-}0340\text{-}verify}\end{array}$$

The $\mathsf{sigHash}\ t$ expression is similar to the definition of sigHashAll from Section 4.5.1. The difference is that we have replaced the $(\mathsf{scribe}(\#^c(\mathsf{hashAll})) \vartriangle \mathsf{hashAll})$ subexpression with $\mathsf{disconnect}\ \mathsf{iden}\ t$. When t is hashAll, these two expressions have identical semantics:

$$\begin{array}{rl} & [\![\mathsf{disconnect\ iden\ hashAll}]\!]^{\mathrm{BC}}\langle\rangle \\ = & ([\![\mathsf{scribe}(\#^c(\mathsf{hashAll})) \vartriangle \mathsf{iden}; \mathsf{iden}; \mathsf{take\ iden} \vartriangle \mathsf{drop\ hashAll}]\!]^{\mathrm{BC}})\langle\rangle \\ = & ([\![\mathsf{take\ iden} \vartriangle \mathsf{drop\ hashAll}]\!]^{\mathrm{BC}} \leftarrowtail \eta^{\mathrm{BC}} \leftarrowtail [\![\mathsf{scribe}(\#^c(\mathsf{hashAll})) \vartriangle \mathsf{iden}]\!]^{\mathrm{BC}})\langle\rangle \\ = & ([\![\mathsf{take\ iden} \vartriangle \mathsf{drop\ hashAll}]\!]^{\mathrm{BC}} \leftarrowtail [\![\mathsf{scribe}(\#^c(\mathsf{hashAll})) \vartriangle \mathsf{iden}]\!]^{\mathrm{BC}})\langle\rangle \\ = & ([\![\mathsf{take\ iden} \vartriangle \mathsf{drop\ hashAll}]\!]^{\mathrm{BC}} \leftarrowtail \eta^{\mathrm{BC}} \circ [\![\mathsf{scribe}(\#^c(\mathsf{hashAll})) \vartriangle \mathsf{iden}]\!])\langle\rangle \\ = & ([\![\mathsf{take\ iden} \vartriangle \mathsf{drop\ hashAll}]\!]^{\mathrm{BC}} \circ [\![\mathsf{scribe}(\#^c(\mathsf{hashAll})) \vartriangle \mathsf{iden}]\!])\langle\rangle \\ = & [\![\mathsf{take\ iden} \vartriangle \mathsf{drop\ hashAll}]\!]^{\mathrm{BC}}\langle [\![\mathsf{scribe}(\#^c(\mathsf{hashAll}))]\!]\langle\rangle, \langle\rangle\rangle \\ = & \phi\langle(\eta^{\mathrm{BC}} \circ [\![\mathsf{scribe}(\#^c(\mathsf{hashAll}))]\!])\langle\rangle, [\![\ \mathsf{hashAll}]\!]^{\mathrm{BC}}\langle\rangle\rangle \\ = & \phi\langle [\![\mathsf{scribe}(\#^c(\mathsf{hashAll}))]\!]^{\mathrm{BC}}\langle\rangle, [\![\ \mathsf{hashAll}]\!]^{\mathrm{BC}}\langle\rangle\rangle \\ = & [\![\mathsf{scribe}(\#^c(\mathsf{hashAll})) \vartriangle \mathsf{hashAll}]\!]^{\mathrm{BC}}\langle\rangle \end{array}$$

The difference between the two expressions is that the subexpression $\mathsf{disconnect\ iden\ hashAll}$ does not commit to using the hashAll transaction digest. During redemption any alternative expression $t : \mathbb{1} \vdash 2^{256}$ for creating a digest can be used instead. Because $\#^c(t)$ is covered by the digital signature proveded to $\mathsf{checkSigHash}\ t$, the signature fixes the digest expression and it cannot be altered by a third party while still satisfying the bip-0340-verify, with the exception of any witness values or disconnected expressions within t. This allows users not only to simulate any of Script's signature hash flags, but to create and use any novel signtature hash mode that they desire.

6.3.1 Side-Effects and Delegation

While the primary purpose of the $t : \mathbb{1} \vdash 2^{256}$ parameter in $\mathsf{checkSigHash}\ t$ is to use the Bitcoin primitives (or more generally the primitives for the specific blockchain application being used) to construct a cryptographic digest, the t parameter can additionally include assertion side-effects. For example, one could include a condition that an absolute or relative timelock meets some specific threshold, without the signature covering any specific timelock value in the digest.

Another possibility is to have the expression return a trivial value (e.g. $\lfloor 0 \rfloor_{256}$) and have t itself contain its own $(\mathsf{checkSigHash}\ t')\langle p', s'\rangle$ program for a different public key and its own digest expression t'. The side-effect of the schnorrAssert within this inner checkSigHash expression essures that the whole Simplicity expression is only successfull if the signature s' is valid for the public key p' and for the transaction digest t'. This process lets you delegate control to another public key by creating a signature with the orginal public key that covers such a delegaton mode for a specific public key, p'. Further conditions can be added as to such a delegation such as checks to limit the amount of funds being spent while ensuring the change is send back to a fixed address. Such delegation can be recursively applied, but could also be restricted by delgating to $(\mathsf{checkSigHashAll})\langle p', s'\rangle$ or to some other fixed set of signature hash modes instead.

Using $(\mathsf{checkSigHash}\ t)\langle p, s\rangle$ as a standard Simplicity expression gives users maximal control over how

they are able to later redeem their funds.

Chapter 7
Type Inference and Serialization

In this chapter we will define a representation of Simplicity expressions as an untyped Simplicity DAG where subexpressions have explicit sharing. We will define how to perform type inference and reconstruct a Simplicity expression from this Simplicity DAG and we will define a binary serialization format for these Simplicity DAGs.

7.1 Explicit Simplicity DAGs

In this section, we will introduce an DAG representation for (untyped) Simplicity expressions with explicit sharing of subexpressions. A Simplicity DAG is a topologically sorted list of Simplicity nodes. Each Simplicity node is a combinator name with a payload of references to earlier nodes in the list, or a payload of witness data, etc.

First we enumerate the possible values for Simplicity nodes.

$$\frac{}{\text{`iden'} : \text{Node}}$$

$$\frac{}{\text{`unit'} : \text{Node}}$$

$$\frac{i : \mathbb{N}}{\text{`injl'}\, i : \text{Node}}$$

$$\frac{i : \mathbb{N}}{\text{`injr'}\, i : \text{Node}}$$

$$\frac{i : \mathbb{N}}{\text{`take'}\, i : \text{Node}}$$

$$\frac{i : \mathbb{N}}{\text{`drop'}\, i : \text{Node}}$$

$$\frac{i : \mathbb{N} \qquad j : \mathbb{N}}{\text{`comp'}\, i\, j : \text{Node}}$$

$$\frac{i : \mathbb{N} \qquad j : \mathbb{N}}{\text{`case'}\, i\, j : \text{Node}}$$

$$\frac{i : \mathbb{N} \qquad j : \mathbb{N}}{\text{`pair'}\, i\, j : \text{Node}}$$

$$\frac{i : \mathbb{N} \qquad j : \mathbb{N}}{\text{`disconnect'}\, i\, j : \mathrm{Node}}$$

$$\frac{v : 2^*}{\text{`witness'}\, v : \mathrm{Node}}$$

$$\frac{b : 2^{512}}{\text{`fail'}\, b : \mathrm{Node}}$$

$$\frac{h : 2^{256}}{\text{`hidden'}\, h : \mathrm{Node}}$$

In addition to the above, every primitive name is a Node. This set of primitives is application specific. For Simplicity with Bitcoin, we have `version' : Node, `lockTime' : Node, etc.

The single quotes around the Node names is there to distinguish them from their corresponding Simplicity combinator. Notice that there is not a perfect 1-to-1 relationship between Node names and Simplicity combinators. In particular, there are no Node names for the assertl and assertr combinators. Instead we have one `hidden' Node name that will be used in conjunction with `case' to represent assertions.

A Simplicity DAG is represented as a topologically sorted, (non-empty) list of Nodes. (You may wish to review Section 2.2.2 to recall our notation for list related operators.)

$$\mathrm{DAG} := \mathrm{Node}^+$$

Each node has between 0 and 2 references to nodes found earlier in the list represented by a relative offset.

$$\begin{aligned}
\mathrm{ref}(\text{`iden'}) &:= \epsilon \\
\mathrm{ref}(\text{`unit'}) &:= \epsilon \\
\mathrm{ref}(\text{`injl'}\, i) &:= i \blacktriangleleft \epsilon \\
\mathrm{ref}(\text{`injr'}\, i) &:= i \blacktriangleleft \epsilon \\
\mathrm{ref}(\text{`take'}\, i) &:= i \blacktriangleleft \epsilon \\
\mathrm{ref}(\text{`drop'}\, i) &:= i \blacktriangleleft \epsilon \\
\mathrm{ref}(\text{`comp'}\, i\, j) &:= i \blacktriangleleft j \blacktriangleleft \epsilon \\
\mathrm{ref}(\text{`case'}\, i\, j) &:= i \blacktriangleleft j \blacktriangleleft \epsilon \\
\mathrm{ref}(\text{`pair'}\, i\, j) &:= i \blacktriangleleft j \blacktriangleleft \epsilon \\
\mathrm{ref}(\text{`disconnect'}\, i\, j) &:= i \blacktriangleleft j \blacktriangleleft \epsilon \\
\mathrm{ref}(\text{`witness'}\, v) &:= \epsilon \\
\mathrm{ref}(\text{`fail'}\, b) &:= \epsilon \\
\mathrm{ref}(\text{`hidden'}\, h) &:= \epsilon \\
\mathrm{ref}(\text{`version'}) &:= \epsilon \\
\mathrm{ref}(\text{`lockTime'}) &:= \epsilon \\
\vdots &\quad \vdots
\end{aligned}$$

In order for a list $l : \mathrm{DAG}$ to be well-formed, it must satisfy two conditions. Firstly, we require that references only have offsets that refer to nodes occurring strictly earlier in the list:

$$\forall \langle k, a \rangle \in \mathrm{indexed}(l).\, \forall i \in \mathrm{ref}(a).\, 0 < i \leq k$$

Secondly, when `hidden' a node is a child, the parent must be a `case' node, which in turn can only have at most one `hidden' child node.

$$\forall \langle k, a \rangle \in \mathrm{indexed}(l).\, \forall i \in \mathrm{ref}(a).\, \forall h.\, l[k-i] = \eta^S(\text{`hidden'}\, h) \Rightarrow \exists\, j_1\, j_2.\, a = \text{`case'}\, j_1\, j_2$$

and

$$\forall \langle k, \text{`case'}\ i\ j\rangle \in \text{indexed}(l).\ \forall h_1 h_2.l[k-i] \neq \eta^S(\text{`hidden'}\ h_1) \vee l[k-j] \neq \eta^S(\text{`hidden'}\ h_2)$$

Theorem 7.1. *If a list l: DAG is well-formed, and l_0: DAG is a prefix of l (i.e. there exists an l_1 such that $\eta^S(l_0)\cdot l_1 = \eta^S(l)$) then l_0 is also well-formed.*

Note that technically the root of a well-formed DAG, $l[|l|-1]$, can be a 'hidden' node.

7.1.1 Type Inference

Simplicity DAGs, as described above, do not have type information associated with them. Before we can interpret a Simplicity DAG as a Simplicity expression we must first perform type inference. Type inference can be done by solving unification equations of typing constraints to compute a most general unifier.

A unification equation is written as $S \doteq T$ were S and T are Simplicity type expressions with unification variables, where unification variables are denoted by Greek letters α, β, γ, etc.

Given a well-formed l: DAG, we associate with each index in the list, $0 \leq k < |l|$, a pair of fresh unification variables α_k, β_k that are to be instantiated at the inferred source and target types of the expression for the node at index k. Each different node occurring at an index k in the DAG l implies a set of unification equations over these type variables, possibly requiring further fresh unification variables.

$$\begin{aligned}
\text{con}_l\langle k, \text{`iden'}\rangle &:= \{\alpha_k \doteq \beta_k\} \\
\text{con}_l\langle k, \text{`unit'}\rangle &:= \{\beta_k \doteq \mathbb{1}\} \\
\text{con}_l\langle k, \text{`injl'}\ i\rangle &:= \{\alpha_k \doteq \alpha_{k-i}, \beta_k \doteq \beta_{k-i} + \gamma\} && \text{where } \gamma \text{ is fresh} \\
\text{con}_l\langle k, \text{`injr'}\ i\rangle &:= \{\alpha_k \doteq \alpha_{k-i}, \beta_k \doteq \gamma + \beta_{k-i}\} && \text{where } \gamma \text{ is fresh} \\
\text{con}_l\langle k, \text{`take'}\ i\rangle &:= \{\alpha_k \doteq \alpha_{k-i} \times \gamma, \beta_k \doteq \beta_{k-i}\} && \text{where } \gamma \text{ is fresh} \\
\text{con}_l\langle k, \text{`drop'}\ i\rangle &:= \{\alpha_k \doteq \gamma \times \alpha_{k-i}, \beta_k \doteq \beta_{k-i}\} && \text{where } \gamma \text{ is fresh} \\
\text{con}_l\langle k, \text{`comp'}\ i\ j\rangle &:= \{\alpha_k \doteq \alpha_{k-i}, \beta_{k-i} \doteq \alpha_{k-j}, \beta_k \doteq \beta_{k-j}\} \\
\text{con}_l\langle k, \text{`case'}\ i\ j\rangle &:= \{\alpha_k \doteq (\gamma_1 + \gamma_2) \times \gamma_3\} \\
&\quad \cup \{\alpha_{k-i} \doteq \gamma_1 \times \gamma_3, \beta_k \doteq \beta_{k-i} | \forall h.l[k-i] \neq \eta^S(\text{`hidden'}\ h)\} \\
&\quad \cup \{a_{k-j} \doteq \gamma_2 \times \gamma_3, \beta_k \doteq \beta_{k-j} | \forall h.l[k-j] \neq \eta^S(\text{`hidden'}\ h)\} \\
& && \text{where } \gamma_1, \gamma_2, \gamma_3 \text{ are fresh} \\
\text{con}_l\langle k, \text{`pair'}\ i\ j\rangle &:= \{\alpha_k \doteq \alpha_{k-i} \doteq \alpha_{k-j}, \beta_k \doteq \beta_{k-i} \times \beta_{k-j}\} \\
\text{con}_l\langle k, \text{`disconnect'}\ i\ j\rangle &:= \{\alpha_{k-i} \doteq \alpha_k \times 2^{256}, \beta_{k-i} \doteq \gamma \times \alpha_{k-j}, \beta_k \doteq \gamma \times \beta_{k-j}\} && \text{where } \gamma \text{ is fresh} \\
\text{con}_l\langle k, \text{`witness'}\ v\rangle &:= \{\} \\
\text{con}_l\langle k, \text{`fail'}\ b\rangle &:= \{\} \\
\text{con}_l\langle k, \text{`hidden'}\ h\rangle &:= \{\} \\
\text{con}_l\langle k, \text{`version'}\rangle &:= \{\alpha_k \doteq \mathbb{1}, \beta_k \doteq 2^{32}\} \\
\text{con}_l\langle k, \text{`lockTime'}\rangle &:= \{\alpha_k \doteq \mathbb{1}, \beta_k \doteq \text{Lock}\} \\
\vdots &\quad \vdots
\end{aligned}$$

The rest of the constraints for the other Bitcoin primitive names follows the same pattern of adding constraints for the α_k and β_k variables to be equal to the input and output types of the corresponding primitives, all of which are required to be concrete types.

Notice that the unification variables for 'hidden' nodes are unused. When 'hidden' nodes occur as children, their parent must be a 'case' node and the constraints for 'case' nodes specifically excludes references to their 'hidden' children's unification variables. Thus when a 'case' node represents an assertion, the only type constraints needed for an assertion are added.

Using the con_l function, we can collect all the constraints that need to be solved for a well-formed (untyped) Simplicity DAG, l: DAG, to be well-typed:

$$\text{con}(l) := \text{fold}^{\cup}(\text{con}_l^{+}(\text{indexed}(l)))$$

Depending on the application there may be further constraints imposed on the root of the DAG. For example, if the DAG is supposed to represent a Simplicity program, which has type $\mathbb{1} \vdash \mathbb{1}$, we would also add the constraints $\{\alpha_{|l|-1} \doteq \mathbb{1}, \beta_{|l|-1} \doteq \mathbb{1}\}$.

A *substitution* ς is a function from unification variables to Simplicity type expressions with unification variables. A substitution, ς, is a *ground substitution* if for every unification variable α, the Simplicity type $\varsigma(\alpha)$ has no unification variables. A substitution, ς, applied to Simplicity type expression S, is a new Simplicity type expression $S|_\varsigma$ with each unification variable α replaced by $\varsigma(\alpha)$:

$$\begin{aligned} \alpha|_\varsigma &:= \varsigma(\alpha) \\ \mathbb{1}|_\varsigma &:= \mathbb{1} \\ (S+T)|_\varsigma &:= S|_\varsigma + T|_\varsigma \\ (S\times T)|_\varsigma &:= S|_\varsigma \times T|_\varsigma \end{aligned}$$

A substitution τ can also be applied to other substitutions yielding a composite substitution:

$$\varsigma|_\tau(\alpha) := \varsigma(\alpha)|_\tau$$

A substitution ς_1 is an instance of another substitution ς_2 whenever there exists a substitution τ such that $\varsigma_1 = \varsigma_2|_\tau$. If ς_1 and ς_2 are both instances of each other then they are α-equivalent.

A unifier for a set of constraints C is a substitution ς such that for every constraint $(S \doteq T) \in C$ we have $S|_\varsigma = T|_\varsigma$. The most general unifier for a set of constraints is a unifier ς for those constraints such that every other unifier ς' for those constraints is an instance of ς. The most general unifier is unique up to α-equivalence.

Once all the constraints have be gathered we can perform first-order unification to solve for the most general unifier ς. The most general unifier, ς, may still contain free variables. To eliminate these free variables, we define an instance of ς that sets all remaining free variable to the unit type $\mathbb{1}$. We call the resulting ground substitution $\varsigma_\mathbb{1}$:

$$\varsigma_\mathbb{1} := \varsigma|_{\lambda\gamma.\mathbb{1}}$$

Notice that if ς and τ are α-equivalent then $\varsigma_\mathbb{1} = \tau_\mathbb{1}$. In particular, this means that the ground substitution $\varsigma_\mathbb{1}$ is independent of which choice of ς we compute as the most general unifier.

It is possible that there is no unifier for the given collection of constraints. In such a case the Simplicity DAG is ill-typed, (or does not meet the type constraints imposed by the application) and does not represent a well-typed Simplicity expression.

First-order unification can be preformed time linear in the size of the constraints [?], although in practice quasi-linear time algorithms using the union-find algorithm are simpler and may perform better on the size of problems one is likely to encounter. Our set of constraints is linear in the size of the DAG thus we can compute the most general unifier in linear (or quasi-linear) time.

It is important to note that these (quasi-)linear time computations rely on having sharing of (type) subexpressions in the representation of the substitution. If you flatten out the representation of the substitution, for example by printing out all the types, the result can be exponential in the size of the input DAG. Fortunately, all of the computation required for Simplicity's consensus operations, can operate without flattening the representation of the inferred types.

Also notice that our set of constraints imply we are doing monomorphic type inference, meaning that any shared subexpressions are assigned the same type. Sometimes we will require multiple instances of the same sub-DAG within a DAG so that different types can be inferred for the subexpressions corresponding to those sub-DAG. As a trivial example, a DAG will often have multiple 'iden' nodes, one for each type that the iden combinator is be used for. A DAG should never have sub-DAGs that end up with the same pair of inferred types and will be disallowed by anti-malleability rules (see Section ⟨reference|TODO⟩).

We could remove duplicate sub-DAGs entirely by using polymorphic type inference instead. Polymorphic type inference is DEXPTIME-complete [?]. However, because removing duplicated sub-DAGs can produce exponentially smaller terms it might be the case that polymorphic type inference is linear in the size of the DAG with duplicated sub-DAGs that are needed for monomorphic type inference. If this is the case, we could switch to polymorphic type inference without opening up DoS attacks. Whether this is possible or not is currently open question for me.

7.1.2 Reconstructing Simplicity Expressions

Given a well-formed Simplicity DAG, l:DAG, that does have a substitution ς yielding the most general unifier for $\mathrm{con}(l)$, we can attempt to synthesize the Simplicity expression that the DAG represents by recursively interpreting the DAG as $\mathrm{syn}(l) : \varsigma_{\mathbb{1}}(\alpha_{|l|-1}) \vdash \varsigma_{\mathbb{1}}(\beta_{|l|-1})$ where

$$\mathrm{syn}(l) := \mathrm{syn}(l, |l|-1, l[|l|-1])$$

and where

$$\begin{aligned}
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`iden'})) &:= \mathsf{iden}_{\varsigma_{\mathbb{1}}(\alpha_k)} \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`unit'})) &:= \mathsf{unit}_{\varsigma_{\mathbb{1}}(\beta_k)} \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`injl'}\, i)) &:= \mathsf{injl}_{\varsigma_{\mathbb{1}}(\alpha_k),B,C}\, \mathrm{syn}(l,k-i,l[k-i]) && \text{where } B+C = \varsigma_{\mathbb{1}}(\beta_k) \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`injr'}\, i)) &:= \mathsf{injr}_{\varsigma_{\mathbb{1}}(\alpha_k),B,C}\, \mathrm{syn}(l,k-i,l[k-i]) && \text{where } B+C = \varsigma_{\mathbb{1}}(\beta_k) \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`take'}\, i)) &:= \mathsf{take}_{A,B,\varsigma_{\mathbb{1}}(\beta_k)}\, \mathrm{syn}(l,k-i,l[k-i]) && \text{where } A\times B = \varsigma_{\mathbb{1}}(\alpha_k) \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`drop'}\, i)) &:= \mathsf{drop}_{A,B,\varsigma_{\mathbb{1}}(\beta_k)}\, \mathrm{syn}(l,k-i,l[k-i]) && \text{where } A\times B = \varsigma_{\mathbb{1}}(\alpha_k) \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`comp'}\, i\, j)) &:= \mathsf{comp}_{\varsigma_{\mathbb{1}}(\alpha_k),\varsigma_{\mathbb{1}}(\beta_{k-i}),\varsigma_{\mathbb{1}}(\beta_k)}\, \mathrm{syn}(l,k-i,l[k-i])\, \mathrm{syn}(l,k-j,l[k-j]) \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`case'}\, i\, j)) &:= \mathrm{syncase}(l,k,k-i,l[k-i],k-j,l[k-j]) \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`pair'}\, i\, j)) &:= \mathsf{pair}_{\varsigma_{\mathbb{1}}(\alpha_k),\varsigma_{\mathbb{1}}(\beta_{k-i}),\varsigma_{\mathbb{1}}(\beta_{k-j})}\, \mathrm{syn}(l,k-i,l[k-i])\, \mathrm{syn}(l,k-j,l[k-j]) \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`disconnect'}\, i\, j)) &:= \mathsf{disconnect}_{\varsigma_{\mathbb{1}}(\alpha_k),B,C,\varsigma_{\mathbb{1}}(\beta_{k-j})}\, \mathrm{syn}(l,k-i,l[k-i])\, \mathrm{syn}(l,k-j,l[k-j]) \\
& && \text{where } B\times C = \varsigma_{\mathbb{1}}(\beta_{k-i}) \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`witness'}\, v)) &:= \mathsf{witness}_{\varsigma_{\mathbb{1}}(\alpha_k),\varsigma_{\mathbb{1}}(\beta_k)}\, \mathrm{inflate}(\varsigma_{\mathbb{1}}(\beta_k), v) \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`fail'}\, b)) &:= \mathsf{fail}_{\varsigma_{\mathbb{1}}(\alpha_k),\varsigma_{\mathbb{1}}(\beta_k)}\, b \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`hidden'}\, h)) &:= \bot \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`version'})) &:= \mathsf{version} \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{`lockTime'})) &:= \mathsf{lockTime} \\
\vdots &\quad \vdots
\end{aligned}$$

The case and witness clauses require special consideration which we define below. The syn function never encounters a 'hidden' node when l is well-formed except when $\exists h. l[|l|-1] = \eta^{\mathrm{S}}(\text{`hidden'}\, h)$, in which case the entire expression itself is hidden. These 'hidden' nodes are only used by the syncase function.

7.1.2.1 syncase

The syncase function constructs case expressions as well as assertl and assertr expressions. Assertion expressions are produced when hidden nodes are passed as parameters. However both branches of a case expression cannot be hidden when l is well-formed.

$$\begin{aligned}
\mathrm{syncase}(l,k_0,k_1,\eta^{\mathrm{S}}(\text{`hidden'}\, h_1),k_2,\eta^{\mathrm{S}}(\text{`hidden'}\, h_2)) &:= \bot \\
\mathrm{syncase}(l,k_0,k_1,n_1,k_2,\eta^{\mathrm{S}}(\text{`hidden'}\, h_2)) &:= \mathsf{assertl}_{A,B,C,\varsigma_{\mathbb{1}}(\beta_{k_0})}\, \mathrm{syn}(l,k_1,n_1)\, h_2 \\
& && \text{where } \varsigma_{\mathbb{1}}(\alpha_{k_0}) = (A+B)\times C \\
& && \text{and when } \forall h_1.\, n_1 \neq \eta^{\mathrm{S}}(\text{`hidden'}\, h_1) \\
\mathrm{syncase}(l,k_0,k_1,\eta^{\mathrm{S}}(\text{`hidden'}\, h_1),k_2,n_2) &:= \mathsf{assertr}_{A,B,C,\varsigma_{\mathbb{1}}(\beta_{k_0})}\, h_1\, \mathrm{syn}(l,k_2,n_2) \\
& && \text{where } \varsigma_{\mathbb{1}}(\alpha_{k_0}) = (A+B)\times C \\
& && \text{and when } \forall h_2.\, n_2 \neq \eta^{\mathrm{S}}(\text{`hidden'}\, h_2) \\
\mathrm{syncase}(l,k_0,k_1,n_1,k_2,n_2) &:= \mathsf{case}_{A,B,C,\varsigma_{\mathbb{1}}(\beta_{k_0})}\, \mathrm{syn}(l,k_1,n_1)\, \mathrm{syn}(l,k_2,n_2) \\
& && \text{where } \varsigma_{\mathbb{1}}(\alpha_{k_0}) = (A+B)\times C \\
& && \text{and when } \forall h_1,h_2.\, n_1 \neq \eta^{\mathrm{S}}(\text{`hidden'}\, h_1) \\
& && \wedge\, n_2 \neq \eta^{\mathrm{S}}(\text{`hidden'}\, h_2)
\end{aligned}$$

7.1.2.2 inflate

A 'witness' node does not hold a Simplicity value, like the witness combinator requires. Instead it has a bit string that encodes a Simplicity value. The inflate function performs a type-directed decoding of this bit string to reconstruct the witness value. The inflate function is defined recursively via the inflation function. These two functions are defined below.

$$\begin{aligned}
\text{inflation}(\mathbb{1}, v) &:= \langle \langle\rangle, v\rangle \\
\text{inflation}(A+B, 0_2 \blacktriangleleft v) &:= \langle \sigma^{\mathbf{L}}_{A,B} a, v'\rangle && \text{where } \langle a, v'\rangle = \text{inflation}(A, v) \\
\text{inflation}(A+B, 1_2 \blacktriangleleft v) &:= \langle \sigma^{\mathbf{R}}_{A,B} b, v'\rangle && \text{where } \langle b, v'\rangle = \text{inflation}(B, v) \\
\text{inflation}(A+B, \epsilon) &:= \bot \\
\text{inflation}(A\times B, v) &:= \langle \langle a, b\rangle, v''\rangle && \text{where } \langle a, v'\rangle = \text{inflation}(A, v) \\
& && \text{and } \langle b, v''\rangle = \text{inflation}(B, v')
\end{aligned}$$

$$\begin{aligned}
\text{inflate}(A, v) &:= a && \text{when } \langle a, \epsilon\rangle = \text{inflation}(A, v) \\
\text{inflate}(A, v) &:= \bot && \text{otherwise}
\end{aligned}$$

As part of DoS protections and malleability protection, we want to prevent witness data from being inflated with unused bits. Notice that that in the definition of $\text{syn}(k, \text{'witness'}\ v)$, the inflate function is passed the inferred type $\varsigma_{\mathbb{1}}(\beta_k)$. Using the inferred type ensures that the witness data only contains data that is nominally useful by the surrounding Simplicity program. Furthermore, the inflate function fails unless it consumes exactly all of its bit string argument. This prevents 'witness' data from being padded with extra, unused bits.

7.2 Serialization

In this section, we define a binary prefix code for serializing Simplicity DAGs. Because prefix codes are self-delimiting, they provide a convenient framework for creating serialization formats. Compound structures can be serialized by concatenation of prefix codes of its substructures.

Our serialization of Simplicity DAGs can be used as part of a network protocol or for writing data to a file. The program size metric (see Section ⟨reference|TODO⟩) computed for terms can be used to bound to the length of this binary encoding. However, specific binary encodings, such of this one, do not form a consensus critical aspect of Simplicity's design and can be substituted with other encodings that have similar suitable bounds. Appendix C describes an alternative, byte-based, binary encoding.

7.2.1 Serialization of Bit Strings and Positive Numbers

In this section we present a recursive Elias prefix code for bit strings and positive natural numbers. Our code for positive natural numbers it has properties similar to the Elias omega coding, but is a has a simple recursive functional definition and some other nice properties.

First, for any $n\colon \mathbb{N}$ with $0 < n$, we define $\lfloor n \rfloor_{2^*}$ to be a bit string for n, written in binary, with the leading 1_2 chopped off.

$$\begin{aligned}
\lfloor 1 \rfloor_{2^*} &:= \texttt{[]}_2 \\
\lfloor 2\,n \rfloor_{2^*} &:= \lfloor n \rfloor_{2^*} \cdot \texttt{[0]}_2 \\
\lfloor 2\,n+1 \rfloor_{2^*} &:= \lfloor n \rfloor_{2^*} \cdot \texttt{[1]}_2
\end{aligned}$$

For example,

$$\begin{aligned}
\lfloor 1 \rfloor_{2^*} &= \texttt{[]}_2 \\
\lfloor 2 \rfloor_{2^*} &= \texttt{[0]}_2 \\
\lfloor 3 \rfloor_{2^*} &= \texttt{[1]}_2 \\
\lfloor 4 \rfloor_{2^*} &= \texttt{[00]}_2 \\
&\vdots \quad \vdots \\
\lfloor 7 \rfloor_{2^*} &= \texttt{[11]}_2 \\
&\vdots \quad \vdots
\end{aligned}$$

This binary code is not a prefix code. To make a prefix code we define mutually recursive function on bit strings and positive numbers. This encodes a bit string by prefixing it with a tag indicating if the bit string is null followed by the bit string's length when it is not null. It encodes a positive number, n, by encoding its associated bit string $\lfloor n \rfloor_{2^*}$.

$$\begin{aligned} \ulcorner \emptyset^{\mathrm{S}}_{2^+} \urcorner &:= \mathtt{[0]}_2 \\ \ulcorner \eta^{\mathrm{S}}_{2^+}(l) \urcorner &:= \mathtt{[1]}_2 \cdot \ulcorner |l| \urcorner \cdot l \\ \ulcorner n \urcorner &:= \ulcorner \lfloor n \rfloor_{2^*} \urcorner \end{aligned}$$

The table below illustrates this prefix code on a few inputs.

$$\begin{array}{rcccl} \ulcorner 1 \urcorner & = & \ulcorner \mathtt{[]}_2 \urcorner & = & \mathtt{[0]}_2 \\ \ulcorner 2 \urcorner & = & \ulcorner \mathtt{[0]}_2 \urcorner & = & \mathtt{[100]}_2 \\ \ulcorner 3 \urcorner & = & \ulcorner \mathtt{[1]}_2 \urcorner & = & \mathtt{[101]}_2 \\ \ulcorner 4 \urcorner & = & \ulcorner \mathtt{[00]}_2 \urcorner & = & \mathtt{[110000]}_2 \\ \ulcorner 5 \urcorner & = & \ulcorner \mathtt{[01]}_2 \urcorner & = & \mathtt{[110001]}_2 \\ \ulcorner 6 \urcorner & = & \ulcorner \mathtt{[10]}_2 \urcorner & = & \mathtt{[110010]}_2 \\ \ulcorner 7 \urcorner & = & \ulcorner \mathtt{[11]}_2 \urcorner & = & \mathtt{[110011]}_2 \\ \ulcorner 8 \urcorner & = & \ulcorner \mathtt{[000]}_2 \urcorner & = & \mathtt{[1101000]}_2 \\ \vdots & & \vdots & & \vdots \\ \ulcorner 15 \urcorner & = & \ulcorner \mathtt{[111]}_2 \urcorner & = & \mathtt{[1101111]}_2 \\ \ulcorner 16 \urcorner & = & \ulcorner \mathtt{[0000]}_2 \urcorner & = & \mathtt{[11100000000]}_2 \\ \vdots & & \vdots & & \vdots \\ \ulcorner 2^{16}-1 \urcorner & = & \ulcorner \mathtt{[1]}_2^{15} \urcorner & = & \mathtt{[11101111]}_2 \cdot \mathtt{[1]}_2^{15} \\ \ulcorner 2^{16} \urcorner & = & \ulcorner \mathtt{[0]}_2^{16} \urcorner & = & \mathtt{[111100000000]}_2 \cdot \mathtt{[0]}_2^{16} \\ \vdots & & \vdots & & \vdots \end{array}$$

Notice that this prefix code preserves numeric ordering as lexicographical ordering. If n and m are positive numbers then $n \leq m \Leftrightarrow \ulcorner n \urcorner \preceq \ulcorner m \urcorner$ where $\preceq$ is the lexicographical ordering on bit strings. When you are parsing a code for a positive number that you know from context is not allowed to exceed some bound b. Then during parsing you can abort as soon as the string being parsed lexicographically exceeds $\ulcorner b \urcorner$. In some cases you can abort parsing after only a few bits. For example, if from context you know that a the positive number must fit in a 64-bit integer (i.e. it is not allowed to exceed $2^{64}-1$), then you can abort parsing as soon as the bit string being parsed lexicographically meets or exceeds the prefix `[1111001]`.

7.2.2 Serialization of Simplicity

In this section we describe a fairly direct serialization for Simplicity DAGs. First, we provide a prefix code for Node values stripped of witness data:

$$\begin{aligned} \ulcorner \text{`comp'}\, i\, j \urcorner &= \mathtt{[00000]}_2 \cdot \ulcorner i \urcorner \cdot \ulcorner j \urcorner \\ \ulcorner \text{`case'}\, i\, j \urcorner &= \mathtt{[00001]}_2 \cdot \ulcorner i \urcorner \cdot \ulcorner j \urcorner \\ \ulcorner \text{`pair'}\, i\, j \urcorner &= \mathtt{[00010]}_2 \cdot \ulcorner i \urcorner \cdot \ulcorner j \urcorner \\ \ulcorner \text{`disconnect'}\, i\, j \urcorner &= \mathtt{[00011]}_2 \cdot \ulcorner i \urcorner \cdot \ulcorner j \urcorner \\ \ulcorner \text{`injl'}\, i \urcorner &= \mathtt{[00100]}_2 \cdot \ulcorner i \urcorner \\ \ulcorner \text{`injr'}\, i \urcorner &= \mathtt{[00101]}_2 \cdot \ulcorner i \urcorner \\ \ulcorner \text{`take'}\, i \urcorner &= \mathtt{[00110]}_2 \cdot \ulcorner i \urcorner \\ \ulcorner \text{`drop'}\, i \urcorner &= \mathtt{[00111]}_2 \cdot \ulcorner i \urcorner \\ \ulcorner \text{`iden'} \urcorner &= \mathtt{[01000]}_2 \\ \ulcorner \text{`unit'} \urcorner &= \mathtt{[01001]}_2 \\ \ulcorner \text{`fail'}\, b \urcorner &= \mathtt{[01010]}_2 \cdot (\mu^* \circ (\iota^{2^8}_{2^*})^* \circ \mathrm{BE})(b) \\ \ulcorner \text{`hidden'}\, h \urcorner &= \mathtt{[0110]}_2 \cdot (\mu^* \circ (\iota^{2^8}_{2^*})^* \circ \mathrm{BE})(h) \\ \ulcorner \text{`witness'} v \urcorner &= \mathtt{[0111]}_2 \end{aligned}$$

We define a witnessData(l) : 2^* function for DAGs l : DAG that serializes the values held by the witness data:

$$\begin{aligned} \text{witnessDatum}(\text{`witness'}\, v) &:= v \\ \text{witnessDatum}(n) &:= \varepsilon \qquad \text{when } \forall v.n \neq \text{`witness'}\, v \end{aligned}$$

$$\text{witnessData}(l) := \ulcorner \mu^*(\text{witnessDatum}^*(\eta^S(l)))\urcorner$$

For serialization of DAGs, l: DAG, which is a non-empty list of Nodes, we have a couple of serialization options.

$$\text{stopCode}(l) := \mu^*((\lambda x.\ulcorner x\urcorner)^*(\eta^S(l)))\cdot \texttt{[01011]}_2 \cdot \text{witnessData}(l)$$

The stopCode above serializes the list of nodes using the prefix code for nodes and terminating the list with the code $\texttt{[01011]}_2$, which has been reserved for use as an end-of-stream marker.

$$\text{lengthCode}(l) := \ulcorner |l| \urcorner \cdot \mu^*((\lambda x.\ulcorner x\urcorner)^*(\eta^S(l)))\cdot \text{witnessData}(l)$$

The lengthCode above prefixes the serialization of DAG with the number of nodes, followed by the list of nodes serialized with the prefix code for nodes. Both stopCode and lengthCode are prefix codes.

The last alternative is to directly use $\mu^*((\lambda x.\ulcorner x\urcorner)^*(\eta^S(l)))$. This is suitable when, from the context of where the code is used, the number of nodes or the length of the code in bits is already known. This variant is not a prefix code.

Which serialization format is best depends on the context in which it is being used. Users should choose the most suitable one for their application.

Notice that the witnessDatum does not use a prefix code, and the witnessData simply concatenates values without separating them with deliminators. This works because during deserialization, type inference does not depend on witness data, and once type inference is complete, we can use the inference types of witness nodes to decode the witness values in sequence. In fact we could even drop the prefix-code use in the witnessData code itself, however we chose to put the block of all the witness values into a prefix-coded list so that one doesn't need to perform type inference to figure out where the the witness data ends.

7.2.2.1 Serialization of Word Jets

A prefix of `[11]` is reserved for computational jets which are given in Appendix B. A previx of `[10]` is used with a special enoding for jetting $\mathsf{scribe}_{\mathbb{1},2^{2^n}}(v)$ expressions, which we call word jets.

$$\frac{n : \mathbb{N} \qquad v : 2^{2^n}}{\text{`word'}\, n\, v : \text{Node}}$$

$$\begin{aligned} \text{ref}(\text{`word'}\, n\, v) &:= \epsilon \\ \text{con}_l\langle k, \text{`word'}\, n\, v\rangle &:= \{\alpha_k \doteq \mathbb{1}, \beta_k \doteq 2^{2^n}\} \\ \text{syn}(l, k, \eta^{\text{S}}(\text{`word'}\, n v)) &:= \mathsf{jet}\langle \mathsf{scribe}_{\mathbb{1},2^{2^n}}(v)\rangle \\ \ulcorner \text{`word'} n v \urcorner &:= \texttt{[10]}_2 \cdot \ulcorner n+1 \urcorner \cdot \iota^{2^{2^n}}_{2^*}(v) \end{aligned}$$

where $\iota^{2^{2^n}}_{2^*}(v)$ formally converts a word value v: 2^{2^n} to a bistring:

$$\begin{aligned} \iota^{2}_{2^*}(b) &:= \eta^*_2(b) \\ \iota^{2^{2^{n+1}}}_{2^*}\langle x, y\rangle &:= \iota^{2^{2^n}}_{2^*}(x)\cdot\iota^{2^{2^n}}_{2^*}(y) \end{aligned}$$

7.2.3 Identity Merkle Root

The DAG encoding for Simplicity expressions allows for sharing of subexpressions, however nothing in the previous section mandates sharing of subexpressions. Without manditory sharing would have a malleability vector where a third party could unshare every subexprssion, vastly inflating the weight of a transaction. To enforce manditory sharing we will introduce the notion of an identity Merkle root and require that every sub-DAG have a unique such identity.

The identity Merkle root of a Simplicity expression t is defined by $\#^i(t)$. This value is the same as $\#^c(t)$ except in the presence of witness and disconnect combinators which are defined as follows:

$$\begin{aligned}\#^i(\mathsf{witness}_{A,B}\, b) &:= \mathrm{SHA256}_{\mathrm{Block}}\big\langle \mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^i_{\mathsf{witness}}}, \langle \mathrm{SHA256}_{\mathbb{2}}(\mathrm{deflate}_B(b)), \#^{\mathrm{ty}}(B)\rangle\big\rangle\\ \#^i(\mathsf{disconnect}_{A,B,C,D}\, s\, t) &:= \mathrm{SHA256}_{\mathrm{Block}}\Big\langle \mathrm{SHA256}_{\mathrm{IV}}^{\mathrm{tag}^i_{\mathsf{disconnect}}}, \langle \#^i(s), \#^i(t)\rangle\Big\rangle\end{aligned}$$

The unique tags used are

$$\begin{aligned}\mathrm{tag}^i_{\mathsf{witness}} &:= \texttt{[53696d706c69636974792d44726166741f4964656e746974791f7769746e657373]}_{\mathbb{2}^8}\\ \mathrm{tag}^i_{\mathsf{disconnect}} &:= \texttt{[53696d706c69636974792d44726166741f4964656e746974791f646973636f6e6e656374]}_{\mathbb{2}^8}\end{aligned}$$

and $\mathrm{deflate}_A : A \to \mathbb{2}^*$ is the inverse of inflate:

$$\begin{aligned}\mathrm{deflate}_{\mathbb{1}}(\langle\rangle) &:= \epsilon_{\mathbb{2}}\\ \mathrm{deflate}_{A+B}(\sigma^{\mathbf{L}}_{A,B}(a)) &:= 0_{\mathbb{2}} \blacktriangleleft \mathrm{deflate}_A(a)\\ \mathrm{deflate}_{A+B}(\sigma^{\mathbf{R}}_{A,B}(b)) &:= 1_{\mathbb{2}} \blacktriangleleft \mathrm{deflate}_B(b)\\ \mathrm{deflate}_{A\times B}(\langle a, b\rangle) &:= \mathrm{deflate}_A(a)\cdot\mathrm{deflate}_B(b)\end{aligned}$$

Lemma 7.2. *For all $a : A$, $\mathrm{inflate}(A, \mathrm{deflate}_A(a)) = a$.*

One of our anti-malleability requirements is that non-hidden nodes in a DAG synethesis are terms with unique identity Merkle root, input, output type triples. We also require that hidden nodes are unique.

Definition 7.3. *We say a well-formed Simplicity DAG $l : \mathrm{DAG}$, that has a substitution ς yielding the most general unifier for $\mathrm{con}(l)$, has maximized sharing when*

1. *for all k_1 and k_2 such that*
   a) *$k_1, k_2 < |l|$ and*
   b) *$l[k_1] \neq \eta^{\mathrm{S}}(\text{`}\mathsf{hidden}\text{'}\ h_1)$ and $l[k_2] \neq \eta^{\mathrm{S}}(\text{`}\mathsf{hidden}\text{'}\ h_2)$ for any h_1 and h_2, and*
   c) *$\varsigma_{\mathbb{1}}(\alpha_{k_1}) = \varsigma_{\mathbb{1}}(\alpha_{k_2})$ and $\varsigma_{\mathbb{1}}(\beta_{k_1}) = \varsigma_{\mathbb{1}}(\beta_{k_2})$, and*
   d) *$\#^i(\mathrm{syn}(l, k_1, l[k_1])) = \#^i(\mathrm{syn}(l, k_2, l[k_2]))$*

   then $k_1 = k_2$, and
2. *for all k_1 and k_2 such that*
   a) *$k_1, k_2 < |l|$ and*
   b) *$l[k_1] = \eta^{\mathrm{S}}(\text{`}\mathsf{hidden}\text{'}\ h)$ and $l[k_2] = \eta^{\mathrm{S}}(\text{`}\mathsf{hidden}\text{'}\ h)$ for some h*

   then $k_1 = k_2$.

Chapter 8
Coq Library Guide

The Coq development for Simplicity is found in the `Coq/` directory. There are two subdirectories: `Simplicity/` contains modules related to Simplicity, and the `Util/` directory has a few modules dedicated to other structures that are not by themselves specific to Simplicity, including a short hierarchy for commutative and idempotent monads. We will focus on the contents of the `Simplicity/` directory.

8.1 Simplicity Types

The Coq development for Simplicity begins with the `Simplicity/Ty.v` file. This contain the inductive definition of `Ty` which defines Simplicity's type expressions. The `tySem` function interprets Simplicity types as Coq types, and it is declared as coercion. The module also provides standard arithmetic notation for Simplicity's sum and product types.

The `tyAlg` is a record collecting the operations needed to define structurally recursive functions for `Ty`. These are known as F-algebras [?], and it is one of the inputs to `tyCata` that defines catamorphisms from `Ty` to another type.

8.2 Simplicity Terms

There are two different ways of representing of Simplicity terms defined in Coq. One representation is an "initial" representation, as an inductive type. The other representation is a "final" representation, based on algebras.

Generally speaking the "initial" representation works well when reasoning about Simplicity term using induction to prove properties about them, while the "final" representation is useful for implicitly capturing shared sub-expressions when defining Simplicity programs.

We begin with the "initial" representation, which will most readers will find more familiar.

8.2.1 The "Initial" Representation of Terms

The `Simplicity/Core.v` module defines an inductive family, `Term A B`, for the well-typed core Simplicity language. The core language is the pure fragment of Simplicity that has no effects such as failure or read access to the transaction environment. The `eval` function provides denotational semantics and interprets terms as Coq functions over the corresponding Coq types.

This module also establishes the core Simplicity completeness theorem (Theorem 3.3) as the `Simplicity_Completeness` theorem. The proof is built from `scribe`, a function to produce Simplicity terms representing constant functions, and `reify` which transforms Coq functions over Simplicity types into Simplicity terms representing those functions.

8.2.2 The "Final" Representation of Terms

To explain the "final" representation of terms it is first necessary to understand what an algebra for Simplicity is. We can understand this by way of a mathematical analogy with ring theory. A ring is a mathematical structure consisting of a domain along with constants from that domain that correspond to 0 and 1, and binary functions over that domain that correspond to $+$ and $\times$ operations that satisfy certain ring laws. A term from the language of rings is an expression made out of 0, 1, $+$, and $\times$. Given a ring and a term from the language of rings, we can interpret that term in the given ring and compute an element of the domain that the term represents. There are many different rings structures, such as the ring of integers, and the ring of integers modulo n for any positive number n. A given term can be interpreted as some value for any ring. An alternative way to represent terms is as a function that, given any ring, returns a value from its domain and does so in a "uniform" way. This would be the "final" representation for terms in the language of rings.

8.2.2.1 Simplicity Algebras

An algebra for Simplicity is an analogous structure to a ring. An algebra for Simplicity consists of a domain, along with constants from that domain that correspond to iden and unit and functions over that domain that correspond the other combinators from Simplicity. Unlike the case for rings, the domain of a Simplicity algebra is indexed by a pair of Simplicity types, and naturally the constants and functions that interpret Simplicity combinators must respect these types (and unlike rings, we are not going to impose any extra laws).

Core Simplicity algebras are formalized in the `Simplicity/Alg.v` file. The `Core.Class.class` record captures the interpretation of constants and combinators for core Simplicity over a given domain. The `Core.Algebra` structure is the type of Simplicity algebras, containing a type family for the domain, and an instance of the `Core.Class.class` record for interpretations.

Given any Simplicity algebra and a well-typed term (from the "initial" representation) we can interpret that term in the algebra to get out a value from the domain (that has a type corresponding to the type of the term). The `Core.eval` function performs this interpretation by recursively evaluating the interpretation of the core Simplicity combinators from the algebra.

What sort of Simplicity algebras are there? The most obvious one is the functional semantics of Simplicity. The domain of this algebra is the functions between Simplicity types. This domain is indexed by the input and output Simplicity types. The interpretation of the iden and unit constants are the identity and constant-unit functions respectively and the interpretation of the other core Simplicity combinators is also in accordance with Simplicity's denotational semantics. This algebra is defined in the `CoreSem` structure and the `CorSem_correct` lemma proves that the interpretation of terms in the "initial" representation into this algebra results in the same function that the `eval` function from `Simplicity/Core.v` produces. The `|[x]|` notation denotes this denotation semantics using the `CoreSem` domain.

Another example of a Simplicity algebra is the "initial" representation of terms themselves, which form a trivial algebra. This domain of Simplicity terms is also indexed by input and output Simplicity types and the constants and combinators are interpreted as themselves. This algebra is defined in the `Core.Term` structure and the `Core.eval_Term` lemma proves that the interpretation of any term in this algebra returns the original term back.

There are several other Simplicity algebras. Programs for the Bit Machine form a Simplicity algebra with the translation from Simplicity to Bit Machine code defining the interpretation of core Simplicity combinators. Also 256-bit hashes form a Simplicity algebra with the commitment Merkle root computation defining the interpretation of core Simplicity combinators. Static analysis of resource usage for Simplicity expressions forms yet another set of Simplicity algebras.

Instances of Simplicity algebras are declared as `Canonical Structures`. This allows Coq's type inference engine to infer the interpretation of Simplicity terms when they are used in the typing contexts of domain of one of these Simplicity algebras.

8.2.2.2 The "Final" Representation

The "final" representation of a Simplicity term is as a function that selects a value out of any Simplicity algebra and does so in a "uniform" manner. A "uniform" manner means a function that satisfies the `Core.Parametric` property which effectively says that that the values chosen by the function from two domains must each be constructed from a composition of the interpretation of combinators in the two domains in the same way. In other words, the function must act the the interpretation of some "initial" represented term under `Core.eval` for any domain.

Terms in the "initial" representation can be converted to the "final" representation by partial application of `Core.eval`. The `Core.eval_Parametric` lemma proves that the resulting "final" representation resulting from `Core.eval` satisfies the `Core.Parametric` property.

Terms in the "final" representation can be converted into the "initial" representation by applying the function to the `Core.Term` Simplicity algebra. The `Core.eval_Term` lemma shows that converting from the "initial" representation to the "final" representation and back to the "initial" representation returns the original value. The `Core.term_eval` lemma shows that starting from any term in the "final" representation that satisfies the `Core.Parametric` property and converting it to the "initial" representation and back to the "final" representation results in an equivalent term. This completes the proof at the two representations are isomorphic.

8.2.2.3 Constructing "Final" Terms

To facilitate the construction of expression in the "final" representation, the nine core combinators are defined as functions parameterized over all Simplicity algebras, and each combinator is proven to be parametric or to preserve parametericity. For the most part, these combinators can be used to write Simplicity expressions in the "final" representation in the same way one would use constructors to write Simplicity expressions in the "initial" representation. On top of this, notation `s &&& t` is defined for the pair combinator, and `s >>> t` is defined for the composition combinator. Also we define the `'H'`, `'O'`, and `'I'` notations for sequences of takes and drops over the identity combinator.

For every expression built in the "final" representation, it is necessary to prove that the result satisfies the parametericity property. A `parametricity` hint database is provided to facilitate automatic proofs of these results. Users should add their own parametricity lemmas to the hint database as they create new Simplicity expressions. Some examples of this can be found in the `Simplicity/Arith.v` module.

8.2.3 Why two representations of Terms?

The "initial" inductive representation is the traditional definition one expects for terms and is easy to reason inductively about. The problem with this representation is that, due to lack of sharing between sub-expressions, it is expensive to evaluate with these terms inside Coq itself. For example, one cannot compute Merkle roots of anything but the most trivial of expressions.

The "final" algebra representation solves this problem by allowing transparent sharing of expressions. In the "final" representation, terms are really values of a Simplicity algebra. When these values are shared using in Coq's let expressions, or shared via some function argument in Coq, those values of the algebra are shared during computation within Coq. This representation makes it feasible to actually compute Merkle roots for Simplicity expressions directly inside Coq.

Both representations are used throughout the Simplicity Coq library. The isomorphism between the two representations is used to transport theorems between them.

I typically use `term : Core.Algebra` as the variable name for an abstract Simplicity algebra. I use this variable name because `Core.Term` are the most generic type of Simplicity algebra (formally known as an initial algebra) so it makes sense to think of generic Simplicity algebras as if they are term algebras.

8.3 Example Simplicity Expressions

8.3.1 Bits

The `Simplicity/Bit.v` file defines notation for the Simplicity type for bits, and notation for their two values `'Bit.zero'` and `'Bit.one'`. The Simplicity expressions `false` and `true` are defined to be the constant functions that return the zero and one bit respectively. A few logical combinators are defined for bits, including the `cond` *`thn els`* combinator which does case analysis on one bit of input, and executes *`thn`* or *`els`* expressions according to whether the bit represented true or false.

All the combinators and Simplicity expressions are given in the "final" representation and parametricity lemmas are provided.

8.3.2 Arithmetic

The `Simplicity/Arith.v` file defines types for multi-bit words and defines Simplicity expressions for addition and multiplication on those words. `Word` *`n`* is a Simplicity type of a 2^n-bit word. The `ToZ` module defines a class of types of finite words. The class provides `toZ` and `fromZ` operations that convert between standard Coq integers and these types of finite words along with proofs that the conversion functions are inverses modulo the word size. `Canonical Structure` declarations provide implementations for the `Bit` and `Word` *`n`* types and for pairs of of such types.

The `Simplicity/Arith.v` file also defines the following Simplicity expressions:

- `adder : forall n term, term (Word n * Word n) (Bit * Word n)`

- `fullAdder : forall n term, term ((Word n * Word n) * Bit) (Bit * Word n)`
- `multiplier : forall n term, term (Word n * Word n) (Word (S n))`
- `fullMultiplier : forall n term,`
  `term ((Word n * Word n) * (Word n * Word n)) (Word (S n))`

The `adder` expression defines the sum of two 2^n-bit word, returning a carry bit and a 2^n-bit word result. The `fullAdder` expression defines the sum of two 2^n-bit word and one (carry input) bit, returning a carry bit and a 2^n-bit word result. The `multiplier` expression defines the product of two 2^n-bit word and returns a 2^{n+1}-bit word. The `fullMultiplier` expression takes a quadruple, $\langle\langle a, b\rangle, \langle c, d\rangle\rangle$ of 2^n-bit words and returns $a \cdot b + c + d$ as a 2^{n+1}-bit word.

Each of these expressions has an associated correctness lemma. These expressions are all defined in the "final" representation and there are parametricity lemmas for each expression.

8.3.3 SHA256

8.4 The Hierarchy of Simplicity Language Extensions

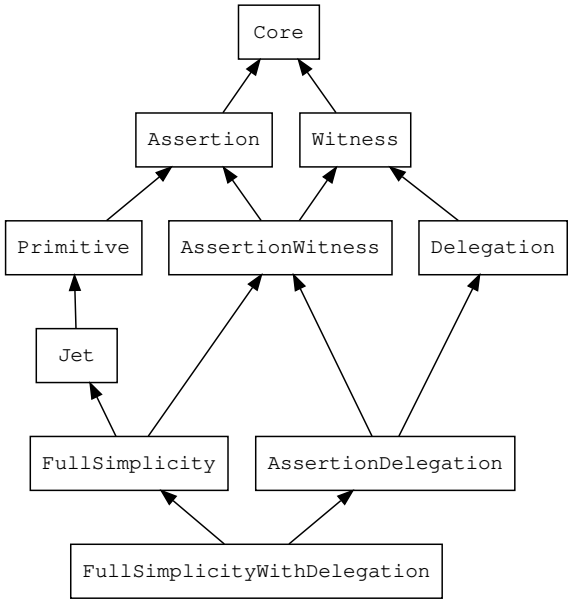

Figure 8.1. The inheritance hierarchy of algebras for Simplicity's partial extensions in Coq.

So far we have only covered the algebra for the core Simplicity language in Coq. The various extensions to this core Simplicity language are captured by extensions to the record type for the core Simplicity algebra. Figure 8.1 illustrates the names of the algebras extending the `Core` language algebra and their inheritance relationship. We use the "packed-classes" method for formalizing the inheritance relation of these extensions in Coq. Readers unfamiliar with this style should first read "Canonical Structures for the working Coq user" [?] and "Packaging Mathematical Structures" [?].

Roughly speaking, there are two dimensions to the inheritance structure. In one direction, the `Core`, `Witness`, and `Delegation` algebras all have semantics that can be interpreted as pure functions. That is, the function semantics of terms from these languages can be evaluated as functions that have no side-effects and can return values within any monad, including the identity monad.

The next layer in this direction, `Assertion`, `AssertionWitness`, and `AssertionDelegation`, extend the previous layer with assertion and failure expressions. These expressions introduce the failure effect and the functional semantics of terms from these languages return values within a `MonadZero`, which includes the `option` monad.

The last layer in this direction, `Primitive`, `Jet`, `FullSimplicity`, and `FullSimplicityWithDelegation`, include primitive terms that are particular to the specific blockchain application. The functional semantics of terms from these language return values within a monad that captures the particular effects of the blockchain application. In the case of Bitcoin, the effects are captured by an environment (a.k.a reader) monad that provides read-only access to the signed transaction data.

We break up the language of Simplicity into these layers because it helps us isolate the side-effects of the various language extensions when reasoning about Simplicity programs. When dealing with a sub-expression from the first layer, one can safely ignore the environment and failure effects and reason only about pure functions. Afterwards, various lemmas, such as `Simplicity.Alg.CoreSem_initial` or `Simplicity.Alg.AssertionSem_initial`, can be used to lift results into the monads use by the other layers when combining the pure sub-expression with other sub-expressions that do have effects.

The other dimension of the inheritance structure breaks the language into feature sets. The `Core`, `Assertion`, and `Primitive` algebras exclude witness and delegation features and encompass the set of language features that `Jet`s are restricted to using. The next layer, `Witness`, `AssertionWitness,` and `FullSimplicity`, add witness features culminating in the `FullSimplcity` algebra defining the Full Simplicity language. The last layer, `Delegation`, `AssertionDelegation`, and `FullSimplicityWithDelgation` provides the powerful and dangerous delegation extension, which should only be used with caution.

We cover these language extensions in more detail below.

8.4.1 Witness

The `Witness` algebra, found in `Simplicity/Alg.v`, extends the `Core` algebra with the `witness` combinator. The `WitnessFunSem` and `WitnessSem` canonical structures define the function semantics by interpreting the terms as pure functions and as Kleisli morphisms for any monad, respectively. The `Witness_initial` lemma relates these two interpretations.

8.4.2 Assertion

The `Assertion` algebra, found in `Simplicity/Alg.v`, extends the `Core` algebra with the `assertl`, `assertr`, and `fail` combinators. The `AssertionSem` canonical structure defines the functional semantics of Simplicity with assertions by interpreting terms as Kleisli morphisms for a monad with zero. The `AssertionSem_initial` lemma shows that when reasoning about the functional semantics of Simplicity with assertions, it suffices to reason within the `option` monad and translate the result to any other monad with zero via the `optionZero` homomorphism.

The `AssertionWitness` algebra is simply the meet of the `Assertion` and `Witness` algebras without adding any new combinators.

8.4.3 Delegation

The `Delegation` algebra, found in `Simplicity/Delegation.v`, extends the `Witness` algebra with the `disconnect` combinator. The `AssertionDelegation` algebra is simply the meet of the `Assertion` and `Delegation` algebras (equiv. the meet of the `AssertionWitness` and `Delegation` algebras) without adding any new combinators.

Building the functional semantics of Simplicity with delegation involves a simultaneous computation of commitment Merkle roots (see Section 8.5) and the functional semantics. To support this the `Delegator arr A B` type is the product of the `arr A B` and the `CommitmentRoot A B` types. Whenever `arr` forms an algebra from any of the previous Simplicity language algebras, then `Delegator arr` is also a member of the same algebra. Furthermore whenever `arr` is a Simplicity with witnesses algebra, then `Delegator arr` is a Simplicity with witnesses and delegation algebra. Similarly whenever `arr` is a Simplicity with assertions and witnesses algebra, then `Delegator arr` is a Simplicity with assertions, witnesses and delegation algebra.

The `runDelegator` projection extracts `arr A B`, from `Delegator arr A B`. For example, `Arrow A B` is a functional semantics for Simplicity with witnesses. Then, when `t` is a term for Simplicity with witnesses and delegation, `runDelegator t : Arrow A B` is the functional semantics of `t`.

The `runDelegator_correctness` lemma shows that for Simplicity terms that don't have delegation, then `runDelegator` returns the original semantics.

8.4.4 Primitives

The Simplicity language is parameterized by the choice of blockchain-specific primitives. Currently we use Coq's module system to capture this parameterization. A `Module Type PrimitiveSig` found in `Simplicity/Primitive.v` defines the parameters that define these blockchain-specific applications of simplicity:

- A type family `t : Ty -> Ty -> Set` of the primitive expression's syntax.
- A function `tag : forall A B, t A B -> hash256` that defines the Merkle roots for the primitives.
- A type `env : Set` that captures the relevant read-only context used to interpret the primitives.
- A function `sem : forall A B, t A B -> A -> env -> option B` that defines the functional semantics of the primitives.

At the moment, this frameworks only supports primitives that use the environment (and failure) side-effect; however this framework could be extended to allow primitives that require other effects that can be captured by commutative and idempotent monads (for example, the writer effect to a commutative and idempotent monoid).

Given an instance of the `PrimitiveSig`'s parameters, the `PrimitiveModule` defines the algebras for the parts of the Simplicity language that depends on the primitives. This includes the `Primitive`, `Jet`, `FullSimplicity` and `FullSimplicityWithDelegation` algebras.

The `Primitive` algebra extends the `Assertion` algebra with the primitives given by the `PrimitiveSig`'s type family `t` through the `prim` combinator. The `primSem M` arrow is the Kleisli arrow for the monad generated by adding an environment effect for the `PrimitiveSig`'s `env` to the monad `M`. The `PrimitivePrimSem` canonical structure provides the functional semantics for Simplicity with primitives by interpreting terms as `primSem M` whenever `M` is a monad zero.

8.4.4.1 Bitcoin

The `Bitcoin` module found in `Simplicity/Primitive/Bitcoin.v` provides these an instance of the `PrimitiveSig` parameters used for a Bitcoin or Bitcoin-like application of Simplicity. The structures defining the signed transaction data are specified culminating in the `sigTx` data type.

The `Bitcoin.prim` type lists the typed primitive expressions defined in Section 4.4.1. The `environment` type captures the read-only context for interpreting these primitives and it includes a `sigTx`, the index withing this transaction that is under consideration, and the commitment Merkle root of the script being evaluated.

Lastly, the `sem` function defines the functional semantics of the primitives in accordance with Section 4.4.1.1 and the `tag` function defines the Merkle roots for the primitives in accordance with Section 4.4.1.2. We use `vm_compute` in the definition of `tag` to pre-evaluate the definitions of the Merkle roots as an optimization.

8.4.5 Jets

The `Jet` algebra, found in the `PrimitiveModule` in `Simplicity/Primtive.v`, extends the `Primitive` algebra with generic support for jets. The `jet` combinator takes a term `t` from the `Primitive` algebra and the `JetPrimSem` canonical structures defines the functional semantics of a jet to be the functional semantics of `t`. Operationally, we expect implementations of specific jets to be natively implemented, but this detail goes beyond the scope of the specification of Simplicity within Coq.

Because `t` is restricted to being a term from the `Primitive` algebra, jets cannot contain `witness` or `disconnect` sub-expressions. While our generic definition of `jets` allows any term from the `Primitive` algebra to be a jet, we expect specific applications of Simplicity to limit themselves to a finite collection of jets through its serialization format.

8.4.6 Full Simplicity

The `FullSimplicity` algebra, found in the `PrimitiveModule` in `Simplicity/Primtive.v`, is the meet of the `Jet` and the `Witness` algebras (equiv. the meet of the `Jet` and `AssertionWitness` algebras) with no additional combinators. It defines the full Simplicity language. The `SimplicityPrimSem` canonical structure provides the functional semantics of the full Simplicity language as the `primSem M` type family when `M` is a monad zero.

The `FullSimplicityWithDelegation` algebra is the the meet of the `Jet` and the `Delegation` algebras (equiv. the meet of the `FullSimplicity` and `Delegation` algebras, or the meet of the `FullSimplicity` and `AssertionDelegation` algebras, etc.) defines the full Simplicity with delegation language. The functional semantics are defined via the `SimplicityDelegationDelegator` canonical structure whose domain includes `Delegator (primSem M)` when `M` is a monad zero. Using `runDelegator`, one can extract a `primSem M` value as the functional semantics.

8.5 Merkle Roots

The `Simplicity/MerkleRoot.v` file defines a Merkle root of types, and the commitment Merkle root and annotated Merkle roots for part of the Simplicity language. The Merkle root of types is specified by `typeRootAlg` and defined by `typeRoot`. The in the `CommitmentRoot A B` family the parameters are phantom parameters, and the value is always a `hash256` type. Canonical Structures provide instances of `CommitmentRoot` for core Simplicity, and Simplicity with assertions and witnesses. The `CommitmentRoot` for delegation is found in `Simplicity/Delegation.v` and the `CommitmentRoot` for primitives, jets, Full Simplicity and Full Simplicity with delegation is found in `Simplicity/Primtive.v`.

These Merkle roots are computed using the SHA-256 compression function with unique tags providing the initial value for each language construct. These tags are in turn the SHA-256 hash of short (less than 56 character) ASCII strings. The Coq definition of SHA-256 is taken from the VST (Verified Software Toolchain) project[?] and the `Simplicity/Digest.v` module provides an interface to that project.

The VST implementation of SHA-256 is efficient enough that it is practical to compute some commitment Merkle roots of functions inside Coq itself using `vm_compute`. See `Fact Hash256_hashBlock` at the end of `Simplicity/SHA256.v` for an example of computing the commitment Merkle root of a Simplicity function that computes the SHA-256 compression function.

8.6 The Bit Machine

The `Simplicity/BitMachine.v` file provides the primary definition of the abstract Bit Machine. This definition, and hence this file, is independent of the rest of the Simplicity language.

The `Cell` type explicitly tracks cell values in the bit machine as being one of `0`, `1`, and undefined. `None` represents the undefined value and `Some false` and `Some true` represent `0` and `1` respectively.

The `ReadFrame` record represents read frames. It uses a zipper representation of a list with a cursor: The elements of the list in front of the cursor are in the `nextData` field and the elements of the list behind the cursor are in the `prevData` field stored in *reverse order*. The `setFrame` function builds a read frame from a list with the cursor set to the beginning of the frame.

The `WriteFrame` record represents write frames. It uses a similar zipper representation where the `writeData` field holds the elements behind the cursor in *reverse order*. Because write frames are append only, every cell in front of the cursor must be an undefined value. For this reason we only store the number of cells in front of the cursor in the `writeEmpty` field. The `newWriteFrame` function builds an empty write frame of a given size and the `fullWriteFrame` function builds an filled write frame from a list.

The `RunState` record represents the non-halted states of the Bit Machine. It consists of

- `inactiveReadFrames`: a list of inactive read frames, with the bottom of the stack at the end of the list.
- `activeReadFrame`: the active read frame, which is the top value of the non-empty stack of read frames.
- `activeWriteFrame`: the active write frame, which is the top value of the non-empty stack of write frames.

- `inactiveWriteFrames`: a list of inactive write frames, with the bottom of the stack at the end of the list.

The `State` variant is either a `RunState` or the `Halted` state and represents the possible states of the Bit Machine. We make the injection of `RunState` into `State` a coercion.

It is sometimes useful to decompose the Bit Machine's state as

$$[\Theta \rhd r_0 \cdot \texttt{[}\underline{c_1 \cdots} \texttt{|}x_{n_0}\texttt{]} \cdot r_0' | w_0 \cdot \texttt{[}c_1 \texttt{|}\cdots \texttt{|}x_{n_1}\texttt{]}\underline{\ } \texttt{[?]}^{n_2} \texttt{[?]}^{m} \lhd \Xi]$$

where we are locally interested in what is immediately in front of the active read frame's cursor, $\texttt{[}\underline{c_1 \cdots} \texttt{|}x_{n_0}\texttt{]}$, and what is immediately surrounding the active write frame's cursor, $\texttt{[}c_1 \texttt{|}\cdots \texttt{|}x_{n_1}\texttt{]}\underline{\ } \texttt{[?]}^{n_2}$. This is captured by the `LocalState` type, noting that the data immediately surrounding the active write frame's cursor is captured by the `WriteFrame` type. The remainder of the state, consisting of $[\Theta \rhd r_0 \cdot \bullet \cdot r_0' | w_0 \cdot \bullet \cdot \texttt{[?]}^{m} \lhd \Xi]$ is captured by the `Context` type, which happens to be isomorphic to the `RunState` type. The `fillContext` function combines a `Context` value and a `LocalState` value to build a complete `RunState` value.

Sometimes we are interested in of some `LocalState` within another `LocalState`. The context of such a decomposition is isomorphic to `LocalState` and we don't even both giving a type alias to this. The `appendLocalState` function combines a context, `ls1`, with a `LocalState`, `ls2`, to build a combined `LocalState`. `appendLocalState` makes `LocalState` into a monoid and `fillContext` becomes a monoid action on `Context`s with respect to this monoid. This theory is not fully developed in Coq, but will be if it is needed. The `context_action` lemma proves the monoid action property, which is the only theorem developed so far.

The `StateShape` type is constructed using similar fields as the `State` type and contains a sequence of numbers. This is used for counting the number of cells in the various components of the Bit Machine's `State`. The `stateShapeSize` function tallies up the totals and is used later in the `maximumMemoryResidence` function.

8.6.1 Bit Machine Code

The `MachineCode.T` *`S1 S2`* type enumerates the nine basic instructions of the Bit Machine. The type of these instructions are parameterized by the legal states that the instructions can successfully operate in, *`S1`*, and the resulting state after execution of the instruction, *`S2`*. In this way, the `MachineCode.T` type family represents a precategory (also known as a directed multi-graph) that captures the machine instructions and their semantics. There is an object (a.k.a. node) for every possible state of the Bit Machine. There is an arrow (a.k.a. directed edge) between two states if there is an instruction of the Bit Machine that successfully transitions from the source state to the target state, and that arrow (a.k.a. directed edge) is labeled with the name of the instruction. The `Abort` instruction is the only instruction whose final state is the `Halted` state. No instruction begins from the `Halted` state. The specific type `MachineCode.T` *`S1 S2`* is the type of all instructions that transition from state *`S1`* to state *`S2`*.

A *thrist* of `MachineCode.T` is the free category generated from the precategory `MachineCode.T`. This free category can also be understood as a collection of all paths through the directed graph of all machine instructions and their associated state transitions. The specific type `Thrst MachineCode.T` *`S1 S2`* is the type of all sequences of instructions that transition from state *`S1`* to state *`S2`*. This type captures the semantics of sequences of machine instructions.

The notation *`S1`* `~~>` *`S2`* denotes the `MachineCode.T` *`S1 S2`* type of single step transitions, which corresponds to $S_1 \rightsquigarrow S_2$. The notation *`S1`* `->>` *`S2`* denotes the `Thrst MachineCode.T` *`S1 S2`* type of multi-step (including 0 steps) transitions between states *`S1`* and `S2` and the trace of the instructions used. The `runHalt` lemma proves that the only trace that begins from the `Halted` state is the empty trace.

8.6.1.1 Bit Machine Programs

We interpret a Bit Machine `Program` as a function taking an initial machine state and, if successful, returning a final machine state along with a thrist of machine instructions that connect the initial state to the final state. The notation `S1 >>- k ->> S2` corresponds to $S_1 \twoheadleftarrow k \twoheadrightarrow S_2$ and denotes that the program `k` when started in state `S1` successfully executes and ends in state `S2`. The `trace` function extracts a `S1 ->> S2` trace from a program `k` when `S1 >>- k ->> S2` holds.

For each machine instruction we use `makeProgram` to define a single instruction `Program` that tries to execute that single instruction once and returns that state transition. If the initial non-halted state given to these single instruction programs is not valid for their instruction, the program fails by returning `None`. However, when the initial state is the `Halted` state, the program succeeds but ignores its instruction and remains in the `Halted` state. This corresponds to the $\boxtimes \rightarrowtail i \twoheadrightarrow \boxtimes$ deduction. These single instruction programs have an associated correctness lemma that proves they run successfully when run from an initial state valid for their instruction and a completeness lemma that proves that they were run from either a valid initial state or the `Halted` state. We also define the trivial `nop` program that contains no instructions and always succeeds.

These single instruction (and `nop`) programs can be combined into more complex programs using the `seq` and `choice` combinators. The `seq` combinator sequences two programs, running the second program starting from the final state of the first program and combines their thrists. The sequence fails if either program fails. The `choice` combinator picks between running two programs by peeking at the cell under the active read frame's cursor from the initial state and running either the first or second program depending on whether the cell holds a `0` or `1` value. When starting from a non-halted state, if the cell holds an undefined value, or if the active read frame's cursor is at the end of the frame, the `choice` combinator fails. When started from the `Halted` state, the `choice` program succeeds but remains in the `Halted` state.

The notations `k0 ;;; k1` and `k0 ||| k1` denote the sequence and choice combinations respectively of two programs and correspond to $k_0; k_1$ and $k_0||k_1$. We also define the combinator `bump n k` which corresponds to $n \star k$.

The `runMachine` function takes a `Program` and an initial `State` and extracts the resulting final `State` and the trace to get there, if the program is successful. For denotational semantics we only care about the resulting final state. For operational semantics we will care how we got there. A few lemmas are provided to help reason about the behaviour of `runMachine` when running the program combinators.

The `maximumMemoryResidence` function computes the maximum number of cells used by any intermediate state from the trace of execution of a Bit Machine program. A few lemmas are provided to help reason about the behaviour of `maximumMemoryResidence` when running the program combinators.

8.6.2 Translating Simplicity to the Bit Machine

The `Simplicity/Translate.v` file defines how to transform Simplicity programs into Bit Machine programs that perform the same computation. The `bitSize` and `encode` functions implement $\mathrm{bitSize}(A)$ and $\ulcorner a \urcorner_A$ respectively.

The `Naive.translate` structure provides a Simplicity algebra for Bit Machine `Progam`s that interprets Simplicity terms according to the naive translation. The `Naive.translate_correct` theorem proves that the `Program` generated by `Naive.translate` when started from a state that contains an encoding of Simplicity function's input successfully ends up in a final machine state that contains an encoding of Simplicity function's output (and input).

8.6.3 Static Analysis

The `Simplicity/StaticAnalysis.v` files defines the static analyses of Simplicity program that compute bounds on the various computational resources used by the Bit Machine when executing translated Simplicity. The file also proves the correctness of these upper bounds.

The `MaximumMemory` module defines the `MaximumMemory.extraMemoryBound` algebra which is used to compute an upper bound on additional memory that will be used when Simplicity sub-expressions are naively translated to the Bit Machine and executed. The `MaximumMemory.Core_spec` lemma proves that for naively translated core Simplicity expressions, the maximum memory used by the Bit Machine is the memory needed by the size of the initial state, plus the results of `MaximumMemory.extraMemoryBound`. This bound holds no matter what the starting state is, even if it is not a valid state for holding the input for the Simplicity expression.

The `MaximumMemory.CellBound` function computes the memory used by the Bit Machine for evaluating Simplicity expressions starting from a standard initial state and `MaximumMemory.CellBound_correct` proves that this upper bound is correct.

Chapter 9
Haskell Library Guide

WARNING: None of the Haskell library development is normative. There is no formalized connection between any of the Haskell library and Simplicity's formal semantics and development in Coq. There could be errors in the Haskell library that cause it to disagree with the formal development defined in Coq. The Haskell library is intended to be used for experimental, exploratory and rapid development of Simplicity related work, but should not be relied upon for production development. For production development, formal developments in Coq should be created.

The Haskell Simplicity project is split up into multiple libraries in order so that multiple different applications of Simplicity to different Blockchains can be developed and used without conflicting with each other. We use the Backpack feature of the Glasgow Haskell Compiler to parameterize and reexport common functionally for different Simplicity applications.

9.1 `Simplicity-Core` library

The first library, `Simplicity-Core`, is found in the `Haskell/Core` directory. It defines the aspects of Simplicity that are independent of any particular Blockchain application. This include Simplicity's type system, Simplicity's core expression language and the assertions, witness, and delegation extensions, and a few other components.

9.1.1 Simplicity Types

The `Core/Simplicity/Ty.hs` file contains the development of Simplicity types. There are three different ways that Simplicity types are captured in Haskell.

The primary way Simplicity types are captured is by the `TyC` class which only has instances for the Haskell types that correspond to the Simplicity types:

- `instance TyC ()`
- `instance (TyC a, TyC b) => TyC (Either a b)`
- `instance (TyC a, TyC b) => TyC (a, b)`

The `TyC` class is crafted so that its methods are not exported. This prevents anyone from adding further instances to the `TyC` class.

The second way Simplicity types are captured is by the `TyReflect` GADT:

```
data TyReflect a where
  OneR  :: TyReflect ()
  SumR  :: (TyC a, TyC b) => TyReflect a -> TyReflect b -> TyReflect (Either a b)
  ProdR :: (TyC a, TyC b) => TyReflect a -> TyReflect b -> TyReflect (a, b)
```

This data type provides a concrete, value-level representation of Simplicity types that is tied to the type-level representation of Simplicity types. For each Haskell type corresponding to a Simplicity type, `a`, the `TyReflect a` type has exactly one value that is built up out of other values of type `TyReflect` corresponding to the Simplicity type sub-expression. For example the value of type `TyReflect (Either () ())` is `SumR OneR OneR`.

The `reify :: TyC a => TyReflect a` produces the value of the `TyReflect` GADT that corresponds to the type constrained by the `TyC` constraint. When users have a Haskell type constrained by `TyC` they can use `reify` to get the corresponding concrete value of the `TyReflect` GADT which can then be further processed. The `reifyProxy` and `reifyArrow` functions are helper functions for `refiy` that let you pass types via a proxy.

The third way Simplicity types are captured is by the `Ty` type alias, which is the fixed point of the `TyF` functor. This is a representation of Simplicity types as a data type. The `one`, `sum`, and `prod` functions provide smart-constructors that handle the explicit fixed-point constructor. The `memoCataTy` helps one build memoized functions that consume `Ty` values.

Generally speaking, we use `TyC` to constrain Haskell types to Simplicity types when creating Simplicity expressions. This way Simplicity type errors are Haskell type errors and can be caught by the Haskell compiler. We use the `Ty` type when doing computations such as deserializing Simplicity expressions and performing unification for Simplicity's type inference. The `TyReflect` GADT links these two representations. For example, the `equalTyReflect` function can test if two Simplicity types are equal or not, and if they are equal then it can unify the Haskell type variables that represent the two Simplicity types. The `unreflect` function turns a `TyReflect` value into a `Ty` value by forgetting about the type parameter.

The `Simplicity/Ty.hs` file also defines the `UntypedValue` that represents data values of Simplicity's typed, but in an untyped manner. This is mostly used for witness nodes. There are functions to convert to and from typed and `UntypedValue`s.

Within the `Simplicity/Ty` directory, there are modules providing data types that are built from Simplicity types. The `Simplicity/Ty/Bit.hs` module provides a `Bit` type, corresponding to 2, and the canonical isomorphism between Haskell's `Bool` type and `Bit`.

The `Simplicity/Ty/Word.hs` module provides the `Word a` type that describes Simplicity types for multi-bit words. Its type parameter is restricted to either be a single `Bit` word type or a product that doubles the size of a another word type via the `Vector` GADT. The `wordSize` returns the number of bits a word has, or more generally the number of elements a vector has. The `fromWord` and `toWord` functions convert values of Simplicity words types to and from Haskell `Integer`s (modulo the size of the word). The file also provides specializations of these various functions for word sizes between 1 and 512 bits.

9.1.2 Simplicity Terms

Terms are represented in tagless-final style [?]. This style is analogous to the "final" representation of terms that is defined in the Coq library.

The `Core/Simplicity/Term/Core.hs` file develops the core Simplicity term language plus a few extensions. The `Core` type class captures Simplicity algebras for core Simplicity expressions. Core Simplicity expressions are represented in Haskell by expressions of type `forall a b. Core term => term a b` which are expressions that hold for all Simplicity algebras.

This module provides infix operators, `(>>>)` and `(&&&)`, for the `comp` and `pair` Simplicity combinators respectively. It also provides notation for short sequences of string of I's, O's and H's. Note that because `case` is a reserved word in Haskell we use `match` for Simplicity's case combinator. Examples of building Simplicity expressions can be found in the next section.

This module also provides `Assert`, `Witness`, and `Delegate` classes for the failure, witness, and delegation language extensions respectively. Terms that make use of these extension will have these class constraints added to their type signatures. For example, a value of type `forall a b. (Core term, Witness term) => term a b` is a term in the language of Simplicity with witnesses.

This module provides `(->)` and `Kleisli m` instances of these classes that provide denotational semantics of core Simplicity and some extensions. For example, one can take core Simplicity terms and directly use them as functions. The semantics of `Delegate` depends on the commitment Merkle root; you can find semantics for that extension in `Indef/Simplicity/Semantics.hs` and it is discussed in Section 9.2.4.

The primary purpose of using tagless-final style is to support transparent sharing of subexpressions in Simplicity. While subexpressions can be shared if we used a GADT to represent Simplicity terms, any recursive function that consumes such a GADT cannot take advantage of that sharing. Sharing results of static analysis between shared sub-expressions is critical to making static analysis practical. Adding explicit sharing to the Simplicity language would make the language more complex and would risk incorrectly implementing the sharing combinator. Explicitly building memoization tables could work, but will have overhead. For instance, we do this when computing Merkle roots of Simplicity types. However, the solution of using tagless-final style lets us write terms in a natural manner and we get sharing for Simplicity expressions at exactly the points where we have sharing in the Haskell representation of the term.

9.1.3 Merkle Roots

The `Core/Simplicity/MerkleRoot.hs` module reexports functionality defined in `Core/Simplicity/MerkleRoot/Impl.hs`, which provides instances of Simplicity terms that compute the commitment, identity and annotated Merkle roots. The `commitmentRoot`, `identityRoot`, and `annotatedRoot` return these Merkle root values. The `Simplicity/MerkleRoot.hs` module also provides a memoized computation of the Merkle roots for Simplicity types.

The SHA-256 implementation is provided through an abstract interface found in `Core/Simplicity/Digest.hs`, which in turn references an implementation of a 256-bit word type defined in `Core/Simplicity/Word.hs`.

The `identityRoot` computation differs from $\#^i(t)$ in that it additionally hashes in the input and output types of an expression t. This is done so that ensuring the uniqueness of the $\#^i(t)$ and the input and outputs type triples can be effectively done by just comparing the single `identityRoot` hash value.

9.1.4 Tensors

The `Simplicity/Tensor.hs` module provides a typed `Product` tensor which allows you to apply multiple interpretations of a Simplicity expression simultaneously. For example, this `Product` is used in the `loop` construct (see Section 9.1.5.5) to simultaneously compute a commitment Merkle root alongside another interpretation.

9.1.5 Example Simplicity Expressions

The `Core/Simplicity/Programs` directory contains various developments of Simplicity expressions in Haskell that are independent of any particular blockchain application.

9.1.5.1 Generic

The `Core/Simplicity/Programs/Generic.hs` file provides some Simplicity expressions that can apply to any Simplicity type.

The `scribe` function produces a Simplicity expression denoting a constant function for any value for any Simplicity type. The `eq` Simplicity expression compares any two values of the same Simplicity type and decides if they are equal or not.

9.1.5.2 Bits

The `Core/Simplicity/Programs/Bit.hs` file has Simplicity expressions for bit manipulation. `false` and `true` are Simplicity expressions for the constant functions of those types and `cond` provides case analysis combinator for a single bit. There are combinators for various logical operators. These logical operators are short-circuited where possible. There are also a few trinary Boolean Simplicity expressions that are used in hash functions such as SHA-256.

9.1.5.3 Multi-bit Words

The `Core/Simplicity/Programs/Word.hs` file provides support for vector an multi-bit word expressions that operate on Simplicity's vector and word types. It provides the implementations of logical bit-vector, padding, truncating, shifting and rotating Simplicity expressions.

The `full_shift` function provides a generic shift operation that allows shifting in and shifting out an arbitary word size of bits. The `shift_const_by` function can perform either left or right shifts, filling in new elements or bits with a provided value. The `rotate_const` function can perform either left or right rotates by a constant amount.

The `bitwise_bin` combinator takes a Simplicity expression for a binary bit operation and lifts it to a Simplicity expression for a binary operation on arbitrary sized words that performs the bit operation bitwise. There is also a variant, called `bitwise_tri` the does the same thing for trinary bit operations.

9.1.5.4 Arithmetic

The `Core/Simplicity/Programs/Arith.hs` file provides support for arithmetic and number based expressions that operate on Simplicity's word types interpreted as unsigned integers. It provides implementations of the `zero`, `one`, `add`, `subtract`, `multiply`, `divide`, and `modulo` Simplicity expressions. It also provides implementations of `lt`, `le`, `min`, `max`, and `median`.

The `eea` expression implements the extended euclidian algorithm, which in turn defines `bezout`, `cofactors`, and `gcd` expressions.

The `absolute_value` and `sign` expression operate on an intepretation of Simplicity words as signed integers (with `absolute_value` returning an unsigned value).

9.1.5.5 Loop

The `Core/Simplicity/Programs/Loop.hs` files is a stub module for holding operations for building unbounded loops via the self-delegation method described in Section 6.2. At the moment it can be used to build the `CommitmentRoot` of an unbounded loop, but the code needed to redeem such a commitment has not been developed yet.

9.1.6 Libraries of Simplicity Expressions

The tagless-final style used for Simplicity expressions is designed to perform efficently for some interpretations of Simplicity when subexpressions are shared. In particular the computation of Merkle roots and serialization (see 9.2.6.2) are much faster to compute when subexpressions are shared. However the polymorphism in the tagless-final type used for expression is at odds with subexpression sharing in a naive implementation of Simplicity expressions.

In order to realize subexpression sharing between Simplicity expressions, we build libraries of Simplicity expressions as a record type. We use the `RecordWildcards` language extension to bring all function of a library in scope with a single `Lib{..}` pattern. This approach is similar to "First-class modules without defaults" by Gabriel Gonzalez. Generating several Simplicity expressions together allows us to share common subexpressions between the different library functions all within a single intepretation.

Libraries typically come with some kind of `mkLib` function that given a set library dependencies, constructs an instance of the library of functions. Building a library from a set of dependencies allows us to share dependencies between different libraries when we want to use multiple different libraries.

The general approach to using the libraries for your own expressions is to create a `where` clause containing the `mkLib` expression for all the libraries that you want to use and their dependencies that are bound to `name@Lib{..}` patterns to bring all the library function into scope. Then you can build your expression using any of the library functions.

9.1.6.1 SHA-256

The `Core/Simplicity/Programs/Sha256.hs` file provides a library of Simplicity expressions to help compute SHA-256 hashes. The `iv` expression is a constant function the returns the initial value to begin a SHA-256 computation. The `hashBlock` expression computes the SHA-256 compression function on a single block of data. To compress multiple blocks, multiple calls to the `hashBlock` function can be chained together.

This library has no dependencies and you can use the `lib` library value directly.

The `Core/Simplicity/Programs/Sha256/Lib.hs` file provides an unpacked module version of the library. However use of this module will lose the subexpression sharing. Therefore this should only be used for testing purposes.

9.1.6.2 LibSecp256k1

The `Core/Simplicity/Programs/LibSecp256k1.hs` file provides a library of Simplicity expressions that mimic the functional behaviour of the the libsecp256k1 elliptic curve library [?]. This includes Simplicity types for, and operations on secp256k1's underlying finite field, elliptic curve point operations in affine and Jacobian coordinates, and linear combinations of points.

This module also include the `bip0340_check` and `bip0340_verify` expressions that implement Schnorr signatures as specified in BIP-0340 [?].

The `mkLib` function builds the library from the its dependency, the SHA-256 library. The `lib` value illustrates how to build the library, but using the `lib` value will not allow you to share the dependency, so it should only be used for testing purposed.

The `Core/Simplicity/Programs/LibSecp256k1/Lib.hs` file provides an unpacked module version of the library. However use of this module will lose the subexpression sharing. Therefore this should only be used for testing purposes.

9.1.6.3 CheckSigHash

The `Core/Simplicity/Programs/CheckSigHash.hs`, while not technically a library, provides a `checkSigHash` Simplicity program that is similiar to the CHECKSIG operation of Bitcoin script except with "Universal signature hash modes" (see Section 6.3). The `sigHash` combinator uses disconnect to pair the commitment root of a given hash mode with its output and produces the message that `checkSigHash` checks its siganture against.

9.1.7 The Bit Machine

The `Core/Simplicity/BitMachine/` directory has modules related to the Bit Machine and evaluation of Simplicity via the Bit Machine.

The `Core/Simplicity/BitMachine/Ty.hs` file defines `bitSize`, `padL`, and `padR`, which define the bitSize, padR and padL functions from Section 3.5.1. They operate on the `Ty` type. The file also defines variants of these three function that operate on the `TyReflect` GADT instead.

The `Core/Simplicity/BitMachine.hs` file (technically not in the `Core/Simplicity/BitMachine/` directory) defines the canonical type of a `Cell` to be a `Maybe Bool`, with the `Nothing` value representing undefined cell values. The `encode` and `decode` functions transform a value of a Simplicity type to and from a list of `Cell`s that represent the value. The `executeUsing` combinator captures a common pattern of running a Simplicity program through an implementation of the Bit Machine by encoding program inputs and decoding the results. Since there is more than one way to compile and run Simplicity program on the Bit Machine (for example, see naive translation versus TCO translation), this abstraction is used is multiple places.

The `MachineCode` type alias captures canonical forms of programs for the Bit Machine, which is the explicit fixed point of the `MachineCodeF` functor. Usually programs are built in continuation passing style (analogous to using difference lists to build lists), making use of the `MachineCodeK` type alias. There are smart-constructors for each machine code that make single instruction `MachineCodeK` programs. Programs are composed sequentially using ordinary function composition, `(.)`. Deterministic choice between two programs is provided by the `(|||)` operator. The `nop` program is an alias for the identity function.

The `Core/Simplicity/BitMachine/Authentic.hs` file is an implementation of the Bit Machine that follows the formal definition of the Bit Machine and fully tracks undefined values. The `Frame` type is used for both read frames and write frames. The `Active` type is captures the pair of active read and write frames, and the `State` type captures the entire state of the Bit Machine. Lenses are used to access the components of the State.

The `runMachine` function interprets `MachineCode` in accordance with the semantics of the Bit Machine, and transforms an initial state into a final state (possibly crashing during execution). It is meant to be used, in conjunction with a Simplicity translator, with `executeUsing`. The `instrumentMachine` function is a variant of `runMachine` that logs statistics about memory usage during the execution. It is used as part of the testing for static analysis.

9.1.7.1 Translating Simplicity to the Bit Machine

The `Core/Simplicity/BitMachine/Translate.hs` file defines the naive translation from Simplicity to the Bit Machine. The `Translation` type wraps the `MachineCodeK` type with phantom type parameters in order to make an instance suitable to be a Simplicity algebra. The `translate` function translates Simplicity terms to `MachineCode` via the `Translation` algebra (recall that a Simplicity term in tagless final form is a polymorphic value that can become any Simplicity algebra). The `Simplicity/BitMachine/Translate/TCO.hs` file provides a similar `Translation` Simplicity algebra and `translate` functions, but this translating using tail composition optimization.

9.1.7.2 Static Analysis

The `Core/Simplicity/BitMachine/StaticAnalysis.hs` file has instances to perform static analysis for bounding the maximum number of cells used by the Bit Machine when executing the naive translation of Simplicity expressions. The `ExtraCellsBnd` type wraps the data needed for the static analysis with phantom type parameters in order to make an instance suitable for a Simplicity Algebra. The `cellsBnd` function computes the bound on cell use from Simplicity terms via the `ExtraCellsBnd` Algebra. The `Core/Simplicity/BitMachine/StaticAnalysis/TCO.hs` file provides a similar static analysis that bounds the maximum number of cells used by the Bit Machine when executing the TCO translation of Simplicity expressions.

The `Core/Simplicity/BitMachine/StaticAnalysis/Tests.hs` runs a few of the example Simplicity expressions through the static analysis and compares the result with the maximum cell count of executing the Bit Machine on various inputs. In this file you can see an example of how `executeUsing (instrumentMachine . translate) program` is used.

9.1.7.3 Fast Evaluation with FFI

While Simplicity with assertions expressions can be evaluated directly with the `Kleisli` instance, evaluation of sophisticated expressions, such as Schnorr signature verification, can be extremely time and memory consuming. To address this the `Core/Simplicity/FFI/Frame.hs` and `Core/Simplicity/FFI/Jets.hs` files create bindings via `cbits/frame.c` and `cbits/jets.c` to the C implementation of jets for Simplicity with assertion expressions. The `Core/Simplicity/CoreJets.hs` modules provides a `fastCoreEval` that replaces evaluation of subexpressions of Simplicity with assertion expressions that match these jets with optimized implementations.

9.2 `Simplicity-Indef` library

To keep the Haskell library of Simplicity modular over different blockchain applications we use the Glasgow Haskell Compiler's Backpack mechanism. The next library, `Simplicity-Indef`, is found in the `Haskell/Indef` directory. The `Indef/Simplicity/Primitive.hsig` file is a module signature that defines the data types and functions that make up the interface that a blockchain application needs to provide. The remained of the `Simplicity-Indef` library defines the aspects of Simplicity that are generic over all different Blockchain application through this `Simplicity.Primitive` module signature. This includes the full Simplicity language, including primitives and jets, the semantics of full Simplicity, type inference for full Simplicity expressions, and generic serialization and deserialization of Simplicity expressions.

Each different blockchain application needs to provide a module satifying the `Simplicity.Primitive` signature. At the moment only the Bitcoin blockchain application is provided (see Section 9.3).

9.2.1 Primitive Signature

The `Indef/Simplicity/Primitive.hsig` signature provides an interface to the different possible primitives provided by different blockchain applications. This signature requires

- `Prim a b`, a GADT for primitives,
- `primPrefix` and `primName` which are used to generate unique names for the Merkle roots of primitive expressions,
- `getPrimBit`, `putPrimBit`, `getPrimByte`, `putPrimByte`, are provide for primitive specific serialization and deserialization operations.
- `PrimEnv` and `primSem`, which provides the type of the context and the denotational semantics for evaluating primitive expressions.

9.2.2 Primitive Terms

The `Indef/Simplicity/Term.hs` module provides expressions for the blockchain primitives and jet extensions, in addition to re-exporting the `Simplicity.Term.Core` module.

Discounted jets are characterized by their specification, which consists of a Simplicity expression with assertions and primitives, but not witness nor delegation. Later a discounted cost will be added as a parameter. Be aware that universal quantifier in the `jet` argument means that subexpressions within this specificaiton cannot be shared outside of the specification itself.

All the Simplicity extensions are gathered together in the `Simplicity` class, whose associated values of type `Simplicity term => term a b` are terms in the full Simplicity language with delegation. The semantics of full Simplicity is discussed in Section 9.2.4.

9.2.3 `JetType` class

While the `Jet` class allows any expression to be a jet, in reality there we will have a specific set of known jets that we have accelerated implements of. The `Indef/Simplicity/JetType.hs` defines the `JetType` class whose instances are types which represent sets of known discounted jets. The `specification` method define the specification of discounted jets and the `matcher` method decides if a given Simplicity expression is known to be substitutable by a some discounted jet. The `implementation` gives an optional optimized implementation of the jet, which otherwise defaults to that of the `specification`. There are also `putJetBit` and `getJetBit` used for Serialization (see Section 9.2.6)

Because the set of discounted jets in use could vary over time, the `JetType` class allows for different types to represent different sets of discounted jets. You can have a different `JetType` instance for each version of the set of discounted jets.

9.2.4 Denotational Semantics of Full Simplicity

The `Indef/Simplicity/Term.hs` module provides an `Kleisli m` instance which provides semantics for full Simplicity (excluding delegation), where `m` is both a reader monad over `PrimEnv` and `MonadFail`. Semantics for the full Simplicity language with delegation, which depends on computing commitment Merkle roots, is found in the `Indef/Simplicity/Semantics.hs` module.

The `Delegator p a b` helper type bundles a commitment Merkle root computation with the Simplicity semantics of `disconnect`, allowing commitment Merkle roots and semantics to be evaluated concurrently. This allows us to create `Delegate` and `Simplicity` instances using `Delegator`.

The `Semantics a b` is a of `Delegator` for the Kleisli semantics that support the Blockchain primitives, and thus is an instance the `Simplicity` class for expressions of full Simplicity with delegation. The `sem` function defines the semantics of full Simplicity with delegation by returning a concrete function from `PrimEnv` and `a` to `Maybe b`.

The `Indef/Simplicity/Semantics.hs` module also provides a `fastEval` function that uses a `JetType`'s optimized `implementation` to evalute subexpressions with known jets.

9.2.5 Type Inference

The file `Indef/Simplicity/Inference.hs` defines a concrete term data type for Simplicity expressions in open recursive style via the `TermF ty j w` functor. The `ty` parameter allows for these terms to be decorated with type annotations, though nothing in the data type itself enforces that the annotations are well-typed. The `tyAnnotation` traversal provides access to the type annotations. When type annotations are unused, this `ty` parameter is set to `()`, as is the case for the `UntypedTermF` functor synonym.

The `j` parameter determines the type of data held by `Jet` nodes. This is usually of the form `SomeArrow arr` for some type annotated data structure `arr` that represents a type for known jets. The `jetData` traversal provides access to the jet data.

The `w` parameter determines the type of data held by `Witness` nodes. This is often `UntypedValue`, but is sometimes a vector or list of `Bool`s, which may be used in intermediate computations when deserializing Simplicity expressions. The `witnessData` combinator is a traversal indexed by type annotations for accessing this `Witness` data.

While the fixed point of this `TermF ty j w` functor would yield a type for untyped, full Simplicity terms, instead we usually use a list or vector of `TermF ty j w Integer` values to build a DAG structure, where the `Integer` values are references to other subexpressions within a list or vector. This provides a structure with explicit sharing of subexpressions. This structure is captured by the `SimplicityDag` type synonym.

The main functions of this module are the `typeInference` and `typeCheck` functions. The normal progression here is to first use the `typeInference` function which discards the type annotations of the input Simplicity DAG (if any) and performs first-order unification to infer new, principle type annotations, with any remaining type variables instantiated at the `()` type. It also adds unification constraints given the input and output types of the intended Simplicity expression provided through the `proxy a b` argument.

Next one would use `traverse . witnessData` to use the inferred type information to decode the witness data into an `UntypedValue`.

Lastly, one would uses the `typeCheck` function to type check the inferred type annotations and, if everything is successful, a proper well-typed Simplicity expression of is returned. Note that the one calling `typeCheck` specifies the type of the resulting Simplicity expression; it is not inferred from the `SimplicityDag`. The `typeCheck` function should never fail after `typeInference` provided the same type constraints in its proxy argument and the witness data has been decoded to an `UntypedValue` that matches the inferred witness type.

There are deserialization functions (see 9.2.6) that go through this progression of type inference and type checking for you.

9.2.6 Serialization

There are two main methods of serialization found in this Simplicity library. The primary method is serialization via a difference list of `Bool`s and deserialization via a free monad representation of a binary branching tree. A difference list, represented within the type `[Bool] -> [Bool]` should be familiar to most Haskell programmers. The same technique is used in the `shows` function using the `ShowS` type synonym and is used to avoid quadratic time complexity in some cases of nested appending of lists. A `DList` type synonym is defined in `Core/Simplicity/Serialization.hs`. Our free monad representation of binary trees is perhaps less familiar. See Section 9.2.6.1 for details.

An alternative serialization method is via the `Get` and `PutM` monads from the `cereal` package. These are used for serializations to and from `ByteStrings`. The alternative method is deprecated and will probably be removed.

9.2.6.1 Free Monadic Deserializaiton

Our free monad representation of binary trees is perhaps less familiar. A binary branching tree with leaves holding values of type `a` can be represented by the free monad over the functor $X \mapsto X^2$, which in Haskell could be written as `Free ((->) Bool) a` where `Free` is from the `Control.Monad.Free` in the `free` package.

```
type BinaryTree a = Free ((->) Bool) a
```

In the free monad representation the `Pure` constructor creates a leaf holding an `a` value while the `Free` constructor builds a branch represented as a function `Bool -> BinaryTree a`, which is isomorphic to a pair of binary trees.

Given a binary tree (represented as a free monad) we can "execute" this monad to produce a value `a` by providing an executable interpretation of each branch. Our interpretation of a branch is to read a bit from a stream of bits and recursively execute either the left branch or the right branch depending on whether we encounter a 0 bit or a 1 bit. This process repeats until we encounter a leaf, after which we halt, returning the value of type `a` held by that leaf. This interpretation captures precisely what it means to use a prefix code to parse a stream of bits. Given a stream of bits we follow branches in a binary tree, left or right, in accordance to the bits that we encounter within the stream until we encounter a leaf, in which case parsing is complete and we have our result, plus a possible remainder of unparsed bits.

Where does the stream of bits come from? Well, if we were to interpret this in the state monad, the state would hold a stream of bits. If we were to interpret this the `IO` monad, we could grab a stream bits from `stdin` or from a file handle. In general, we can interpret our free monad in any monad that offers a callback to generate bits for us to consume.

```
runBinaryTree : Monad m => m Bool -> BinaryTree a -> m a
runBinaryTree next = foldFree (<$> next)
```

This ability to interpret a free monad within any other monad is essentially what it means to be a free monad in the first place.

This free monad approach to parsing can be extended. For example, suppose when parsing we encounter a prefix that is not a code of any value. We can extend our functor to given a variant to return failure in this case. Thus we build a free monad over the functor $X \mapsto 1 + X^2$. In Haskell we could use the type `Free (Sum (Const ()) ((->) Bool))` for this monad or equivalently

```
type BitDecoder a = Free (Sum (Const ()) ((->) Bool)) a

runBitDecoder : Monad m => m Void -> m Bool -> BitDecoder a -> m a
runBitDecoder abort next = foldFree eta
 where
  eta (Inl (Const ())) = vacuous abort
  eta (Inr f) = f <$> next
```

Our free monad interpreter now requires two callbacks. The `next` callback is as before; it generates bits to be parsed. The `abort` callback handles a failure case when the a sequence of bits do not correspond to any coded value. This callback can throw an exception or call `fail`, or do whatever is appropriate in case of failure.

This implementation of free monads suffers from a similar quadratic complexity issue that lists have. In some cases, nested calls to the free monad's bind operation can have quadratic time complexity. To mitigate this we choose a to use a different representation of free monads.

The above interpreters completely characterize their corresponding free monads. Instead of using the `BinaryTree a` type we can directly use the type `forall m. Monad m => m Bool -> m a`. Similarly we can directly use the type `forall m. Monad m => m Void -> m Bool -> m a` in place of the `BitParser a` type TODO: consider replacing `m Void` with a `MonadFail` constraint instead. This is known as the Van Laarhoven free monad representation [?] and it is what we use in this library.

For example, `getBitString` and `getPositive` from the `Simplicity.Serialization` module are decoders a list of bits and positive numbers respectively that use this Van Laarhoven representation of binary trees. Similarly `get256Bits` from `Simplicity.Digest` is a decoder for a 256-bit hash value.

In `Core/Simplicity/Serialization.hs` there are several adapter functions for executing these Van Laarhoven free monads within particular monads.

- `evalStream` evaluates a Van Laarhoven binary tree using a list of bits and returns `Nothing` if all the bits are consumed before decoding is successful.
- `evalExactVector` evaluates a Van Laarhoven binary tree using a vector of bits and will return `Nothing` unless the vector is exactly entirely consumed.
- `evalStreamWithError` evaluates a Van Laarhoven bit decoder using a list of bits and returns an `Error` if the decoder aborts or the list runs out of bits.
- `getEvalBitStream` evaluates a Van Laarhoven bit decoder within cereal's `Get` monad while internally tracking partially consumed bytes.

9.2.6.2 Serialization of Simplicity DAGs

The file `Indef/Simplicity/Dag.hs` provides a `jetDag` that coverts Simplicity expressions into a topologically sorted DAG structure with explicit sharing that is suitable for encoding. This conversion

- finds and shares identical well-typed subexpressions,
- runs type inference to determine the principle type annotations needed to optimal sharing and pruning of unused witness data,

- finds subexpressions that matches known jets and replaces them with `Jet` nodes.

The file `Indef/Simplicity/Serialization/BitString.hs` provides `getTermLengthCode` and `putTermLengthCode` functions that decode and encode a Simplicity expression. The `putTermLengthCode` function executes `jetDag` to perform sharing and substitution of jets, and the `getTermLengthCode` executes deserialization, type inference and type checking all together. The `getTermStopCode` and `putTermStopCode` functions provide the same functionality using a serialization format with a stop code instead of a length code prefix. The module also provides, `getDagNoWitness`, `getWitnessData` and `putDag`, that are used by the `getTerm*` and `putTerm*` functions to convert between Simplicity DAGs and their serialized representation.

The file `Indef/Simplicity/Serialization/ByteString.hs` provides similar `getDag` and `putDag` functions for the alternative `ByteString` encoding described in Appendix C.

9.2.7 Jet Substitution

The `Indef/Simplicity/Dag.hs` provides also provides a `jetSubst` function uses the same substitution mechanism in found in Section 9.2.6.2 to return a new Simplicity expression that replaces found jets with explicit jets. This substitution can change the commitment root of the overall expression and of subexpressions. Because the commitment root is used in the semantics of `disconnect`, it is even possible for this substitution to change the result of evaluation of expressions.

In order to support partial evaluation, the `jetSubt` function returns a `WrappedSimplicity` expression. This result can be unwrapped by the `unwrap` function. The use of `WrappedSimplicity` lets us share much of the work of substitution in case we want to evaluate the resulting expression with multiple different interpretations. Without `WrappedSimplicity` we would end up needing to redo the entire subtitution for each different intepretation of the resulting expression.

9.3 `Simplicity-Bitcoin` Libary

To instantiate the `Simplicity-Indef` library, we need to provide a blockchain specific implementation of the `Simplicity.Primitive` signature. The `Simplicity-Bitcoin` library provides primitives used for Bitcoin applications. The `Bitcoin/Simplicity/Bitcoin/Primitive.hs` module implements the `Simplicity.Primitive` signature by providing the primitive expressions and their semantics for Simplicity's Bitcoin application. The `Prim a b` GADT enumerates the list of primitive Simplicity expressions for Bitcoin. The `PrimEnv` provides the context that a Simplicity expression is evaluated within, providing the signed transaction data, the index of the input being considered for redemption, and the commitment Merkle root of the Simplicity program itself. The `primSem` function is an interpreter for these primitive expressions for the Bitcoin.

The `Bitcoin/Simplicity/Bitcoin/DataTypes.hs` module provides the data structures that make up the signed transaction data for Bitcoin.

9.4 `Simplicity` Library

The `Simplicty` library assembles all of the previous libraries together. The `Simplicity-Indef` library is instantiated at all available implementations of the `Simplicity.Primitive` signature, which at the moment is only the `Simplicity.Bitcoin.Primitive` module. This Bitcoin instance of the `Simplicity-Indef` library has its modules reexported under the `Simplicity.Bitcoin` prefix. Specifically the following modules are reexported:

- `Simplicity.Term` as `Simplicity.Bitcoin.Term`
- `Simplicity.Semantics` as `Simplicity.Bitcoin.Semantics`
- `Simplicity.Dag` as `Simplicity.Bitcoin.Dag`

- `Simplicity.Inference` as `Simplicity.Bitcoin.Inference`
- `Simplicity.Serialization.BitString` as `Simplicity.Bitcoin.Serialization.BitString`
- `Simplicity.Serialization.ByteString` as `Simplicity.Bitcoin.Serialization.ByteString`

9.4.1 CheckSigHashAll

Some modules build on specific Simplicity blockchain applications. The `Simplicity/Bitcoin/Programs/CheckSigHashAll.hs` file provides a library for a `hashAll` signature hash mode for creating a Bitcoin specific transaction digest and a `checkSigHashAll` Simplicity program that verifies Schnorr signature over that digest for a given public key. The `sigHashAll` function hashes the pair of the commitment root of `hashAll` with the output of `hashAll` producing the message that needs to be signed. This signature is compatible with using the `hashAll` mode with the `checkSigHash` program from Section 9.1.6.3.

The `Simplicity/Bitcoin/Programs/CheckSigHashAll/Lib.hs` file provides an unpacked module version of the library. However use of this module will lose the subexpression sharing. Therefore this should only be used for testing purposes.

The `Simplicity/Elements/Programs/CheckSigHashAll.hs` file provides similar functionality for the Elements blockchain application.

9.4.2 Known Discounted Jets

The `Simplicity.Bitcoin.Jets` and `Simplicity.Elements.Jets` provide a canonical `JetType` of known jets. Currently this only consists of `CoreJets`. These modules also provide `getTerm*` and `putTerm*` function that specifically encode and decode this set of jets.

Both sets of jets draw upon the `Simplicity.CoreJets` module which provides "core" jets that are jets that do not depend on any primitives. These "core" jets include

- Jets for arithmetic, including 32-bit addition, subtraction and multiplication
- Jets for hash functions, including the SHA-256 compression function.

These modules also reexport specialized instances of `jetSubst` and `fastEval` for their specific `JetTypes`.

9.5 Simplicity `testsuite`

The `Tests` directory has a collection of tests for a Simplicity `testsuite`. The `Tests/Tests.hs` file imports the various test modules to build a testing executable to run them all.

The `Tests/Simplicity/Programs/Tests.hs` has some QuickCheck properties that provide randomized testing for some of the Simplicity expressions developed.

The `Tests/Simplicity/BitMachine/Tests.hs` runs a few of the Simplicity expressions through the Bit Machine implementation to test that the value computed by the Bit Machine matches that direct interpretation of the same Simplicity expressions. In this file you can see an example of how `executeUsing`

`(runMachine . translate) program` is used.

Chapter 10
C Library Guide

Appendix A
Elements Application

The Elements application of Simplicity is based on the Bitcoin application described in Section 4.4.1. The signed transaction data for Elements is similar to Bitcoin's but with added confidential amounts and assets, pegins, and asset issuances. Below we define the record type ELEnv defining the environment in which Simplicity expressions are evaluated within for the Elements application.

$$
\begin{aligned}
\text{Lock} &:= 2^{32}\\
\text{Outpoint} &:= 2^{256} \times 2^{32}\\
\text{Confidential}(X) &:= \left\{ \begin{array}{l} \text{pt} : \text{Point} \\ \text{prf} : X \end{array} \right\}\\
\text{ExplicitAsset} &:= 2^{256}\\
\text{Asset}(X) &:= \text{Confidential}(X) + \text{ExplicitAsset}\\
\text{ExplicitAmount} &:= 2^{64}\\
\text{Amount}(X) &:= \text{Confidential}\,(X) + \text{ExplicitAmount}\\
\text{ExplicitNonce} &:= 2^{256}\\
\text{Nonce} &:= \text{Confidential}\,(\mathbb{1}) + \text{ExplicitNonce}\\
\text{RangeProof} &:= (2^8)^*\\
\text{SurjectionProof} &:= (2^8)^*\\
\text{TokenAmount}(X) &:= \text{Amount}(X)\\
\text{NewIssuance} &:= \left\{ \begin{array}{l} \text{contractHash} : 2^{256} \\ \text{amount} : \text{Amount}(\text{RangeProof}) \\ \text{tokenAmount: TokenAmount}(\text{RangeRpoof}) \end{array} \right\}\\
\text{Reissuance} &:= \left\{ \begin{array}{l} \text{blindingNonce} : \text{ExplicitNonce} \\ \text{entropy} : 2^{256} \\ \text{amount} : \text{Amount}(\text{RangeProof}) \end{array} \right\}\\
\text{Issuance} &:= \text{NewIssuance} + \text{Reissuance}\\
\text{ElementsSigOutput} &:= \left\{ \begin{array}{l} \text{asset} : \text{Asset}(\text{SurjectionProof}) \\ \text{amount} : \text{Amount}(\text{RangeProof}) \\ \text{scriptPubKey} : (2^8)^* \\ \text{nonce} : \text{S}(\text{Nonce}) \end{array} \right\}\\
\text{ElementsUTXO} &:= \left\{ \begin{array}{l} \text{asset} : \text{Asset}(\mathbb{1}) \\ \text{amount} : \text{Amount}(\mathbb{1}) \\ \text{scriptPubKey} : (2^8)^* \end{array} \right\}\\
\text{ElementsSigInput} &:= \left\{ \begin{array}{l} \text{peginChain} : \text{S}(2^{256}) \\ \text{prevOutpoint} : \text{Outpoint} \\ \text{txo} : \text{ElementsUTXO scriptPubKey is claim\_script when isPegin} \\ \text{sequence} : 2^{32} \\ \text{issuance: S}(\text{Issuance}) \\ \text{annex} : \text{S}((2^8)^*)\ \text{(excludes the } \mathtt{50}_{2^8} \text{ prefix)} \\ \text{scriptSig} : (2^8)^* \end{array} \right\}
\end{aligned}
$$

$$\text{ElementsSigTx} := \left\{ \begin{array}{l} \text{version} : 2^{32} \\ \text{inputs} : \text{ElementsSigInput}^{+} \\ \text{outputs} : \text{ElementsSigOutput}^{+} \\ \text{lockTime} : \text{Lock} \end{array} \right\}$$

$$\text{ELEnv} := \left\{ \begin{array}{l} \text{tx} : \text{ElementsSigTx} \\ \text{ix} : 2^{32} \\ \text{tapEnv} : \text{TapEnv} \\ \text{genesisBlockHash} : 2^{256} \\ \text{scriptCMR} : 2^{256} \end{array} \right\}$$

The parameterized types of Confidential, Asset, and Amount are functors in their natural way.

$$\begin{array}{rcl} (\text{Confidential}(f)(a))[\text{pt}] & := & a[\text{pt}] \\ (\text{Confidential}(f)(a))[\text{prf}] & := & f(a[\text{prf}]) \\ \text{Asset}(f)(\sigma^{\mathbf{L}}(a)) & := & \sigma^{\mathbf{L}}(\text{Confidential}(f)(a)) \\ \text{Asset}(f)(\sigma^{\mathbf{R}}(a)) & := & \sigma^{\mathbf{R}}(a) \\ \text{Amount}(f)(\sigma^{\mathbf{L}}(a)) & := & \sigma^{\mathbf{L}}(\text{Confidential}(f)(a)) \\ \text{Amount}(f)(\sigma^{\mathbf{R}}(a)) & := & \sigma^{\mathbf{R}}(a) \end{array}$$

The Elements protocol imposes limits and other constraints similar to Bitcoin's for transactions to be valid. We will assume that for all transactions the length of the inputs and outputs arrays are less than or equal to 2^{25}, and the the length of the scriptPubKeys lengths are also less than or equal to 2^{25}. Also we will assume that for every $e : \text{ELEnv}$ that $\lceil e[\text{ix}] \rceil < |e[\text{tx}][\text{inputs}]|$ so that "current" index being validated is, in fact, an input of the transaction.

The monad we use for the Elements application provides an environment effect (also known as a reader effect) that allows read-access to the ELEnv value defining the Simplicity program's evaluation context. We call this monad EL.

$$\begin{array}{rcl} \text{EL}A & := & \text{ELEnv} \to \text{S}A \\ \text{EL}f(a) & := & \lambda e : \text{ELEnv}.\, \text{S}f(a(e)) \end{array}$$

EL is a commutative, idempotent monad with zero:

$$\begin{array}{rcl} \eta^{\text{EL}}_A(a) & := & \lambda e : \text{ELEnv}.\, \eta^{\text{S}}_A(a) \\ \mu^{\text{EL}}_A(a) & := & \lambda e : \text{ELEnv}.\, \mu^{\text{S}}_A(\text{S}(\lambda f.\, f(e))(a(e))) \\ \emptyset^{\text{EL}}_A & := & \lambda e : \text{ELEnv}.\, \emptyset^{\text{S}}_A \end{array}$$

We define several new primitive expressions for reading data from a ELEnv value. The language that uses this extension is called *Simplicity with Elements*.

$$\frac{}{\mathsf{version} : \mathbb{1} \vdash 2^{32}}$$

$$\frac{}{\mathsf{lockTime} : \mathbb{1} \vdash \text{Lock}}$$

$$\frac{}{\mathsf{inputPegin} : 2^{32} \vdash \mathrm{S}(\mathrm{S}(2^{256}))}$$

$$\frac{}{\mathsf{inputPrevOutpoint} : 2^{32} \vdash \mathrm{S}(\mathrm{Outpoint})}$$

$$\frac{}{\mathsf{inputAsset} : 2^{32} \vdash \mathrm{S}(\mathrm{Asset}(\mathbb{1}))}$$

$$\frac{}{\mathsf{inputAmount} : 2^{32} \vdash \mathrm{S}(\mathrm{Amount}(\mathbb{1}))}$$

$$\frac{}{\mathsf{inputScriptHash} : 2^{32} \vdash \mathrm{S}(2^{256})}$$

$$\frac{}{\mathsf{inputSequence} : 2^{32} \vdash \mathrm{S}(2^{32})}$$

$$\frac{}{\mathsf{inputAnnexHash} : 2^{32} \vdash \mathrm{S}(\mathrm{S}(2^{256}))}$$

$$\frac{}{\mathsf{inputScriptSigHash} : 2^{32} \vdash \mathrm{S}(2^{256})}$$

$$\frac{}{\mathsf{reissuanceBlinding} : 2^{32} \vdash \mathrm{S}(\mathrm{S}(\mathrm{ExplicitNonce}))}$$

$$\frac{}{\mathsf{newIssuanceContract} : 2^{32} \vdash \mathrm{S}(\mathrm{S}(2^{256}))}$$

$$\frac{}{\mathsf{reissuanceEntropy} : 2^{32} \vdash \mathrm{S}(\mathrm{S}(2^{256}))}$$

$$\frac{}{\mathsf{issuanceAssetAmount} : 2^{32} \vdash \mathrm{S}(\mathrm{S}(\mathrm{Amount}(\mathbb{1})))}$$

$$\frac{}{\mathsf{issuanceTokenAmount} : 2^{32} \vdash \mathrm{S}(\mathrm{S}(\mathrm{TokenAmount}(\mathbb{1})))}$$

$$\frac{}{\mathsf{issuanceAssetProof} : 2^{32} \vdash \mathrm{S}(2^{256})}$$

$$\frac{}{\mathsf{issuanceTokenProof} : 2^{32} \vdash \mathrm{S}(2^{256})}$$

$$\frac{}{\mathsf{currentIndex} : \mathbb{1} \vdash 2^{32}}$$

$$\frac{}{\mathsf{outputAsset} : 2^{32} \vdash \mathrm{S}(\mathrm{Asset}(\mathbb{1}))}$$

$$\frac{}{\mathsf{outputAmount} : 2^{32} \vdash \mathrm{S}(\mathrm{Amount}(\mathbb{1}))}$$

$$\frac{}{\mathsf{outputNonce} : 2^{32} \vdash \mathrm{S}(\mathrm{S}(\mathrm{Nonce}))}$$

$$\frac{}{\mathsf{outputScriptHash} : 2^{32} \vdash \mathrm{S}(2^{256})}$$

$$\frac{}{\mathsf{outputNullDatum} : 2^{32} \times 2^{32} \vdash \mathrm{S}(\mathrm{S}(2^2 \times 2^{256} + (2 + 2^4)))}$$

$$\frac{}{\mathsf{outputSurjectionProof} : 2^{32} \vdash \mathrm{S}(2^{256})}$$

$$\frac{}{\mathsf{outputRangeProof} : 2^{32} \vdash \mathrm{S}(2^{256})}$$

$$\frac{}{\mathsf{tapleafVersion} : \mathbb{1} \vdash 2^{8}}$$

$$\frac{}{\mathsf{tapbranch} : 2^{8} \vdash \mathrm{S}(2^{256})}$$

$$\frac{}{\mathsf{internalKey} : \mathbb{1} \vdash \mathrm{PubKey}}$$

$$\frac{}{\mathsf{genesisBlockHash} : \mathbb{1} \vdash 2^{256}}$$

$$\frac{}{\mathsf{scriptCMR} : \mathbb{1} \vdash 2^{256}}$$

A.1 Denotational Semantics

We extend the formal semantics of these new expressions as follows.

$$\begin{aligned}
[\![\mathsf{version}]\!]^{\mathrm{EL}}\langle\rangle &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{tx}][\mathrm{version}]) \\
[\![\mathsf{lockTime}]\!]^{\mathrm{EL}}\langle\rangle &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{tx}][\mathrm{lockTime}]) \\
[\![\mathsf{inputPegin}]\!]^{\mathrm{EL}}\langle\rangle &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, l[\mathrm{peginChain}])(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{inputPrevOutpoint}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, l[\mathrm{prevOutpoint}])(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{inputAsset}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, l[\mathrm{txo}][\mathrm{asset}])(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{inputAmount}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, l[\mathrm{txo}][\mathrm{amount}])(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{inputScriptHash}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, \mathrm{SHA256}(l[\mathrm{txo}][\mathrm{scriptPubKey}]))(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{inputSequence}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, l[\mathrm{sequence}])(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{inputAnnexHash}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, \mathrm{S}(\mathrm{SHA256})(l[\mathrm{annex}]))(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{inputScriptSigHash}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, \mathrm{SHA256}(l[\mathrm{scriptSig}]))(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{reissuanceBlinding}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, \mathrm{S}(\mathrm{blindingNonce})(\mathrm{reissuance}(l))) \\
&\qquad (e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{newIssuanceContract}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, \mathrm{S}(\mathrm{contractHash})(\mathrm{newIssuance}(l))) \\
&\qquad (e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{reissuanceEntropy}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\, \mathrm{S}(\mathrm{entropy})(\mathrm{reissuance}(l)))(e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{issuanceAssetAmount}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\mathrm{S}(\mathrm{ClearAmout}) \circ \mathrm{issuanceAssetAmt}) \\
&\qquad (e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{issuanceTokenAmount}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\mathrm{S}(\mathrm{ClearAmout}) \circ \mathrm{issuanceTokenAmt}) \\
&\qquad (e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{issuanceAssetProof}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\mathrm{OptionPrfHash} \circ \mathrm{issuanceAssetAmt}) \\
&\qquad (e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{issuanceTokenProof}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\mathrm{OptionPrfHash} \circ \mathrm{issuanceTokenAmt}) \\
&\qquad (e[\mathrm{tx}][\mathrm{inputs}]\lceil i \rceil)) \\
[\![\mathsf{currentIndex}]\!]^{\mathrm{EL}}\langle\rangle &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{ix}])
\end{aligned}$$

$$
\begin{aligned}
[\![\mathsf{outputAsset}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\ \mathrm{ClearAsset}(l[\mathrm{asset}]))(e[\mathrm{tx}][\mathrm{outputs}]\lceil i \rceil)) \\
[\![\mathsf{outputAmount}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\ \mathrm{ClearAmount}(l[\mathrm{amount}]))(e[\mathrm{tx}][\mathrm{outputs}]\lceil i \rceil)) \\
[\![\mathsf{outputScriptHash}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\ \mathrm{SHA256}(l[\mathrm{scriptPubKey}]))(e[\mathrm{tx}][\mathrm{outputs}]\lceil i \rceil)) \\
[\![\mathsf{outputNullDatum}]\!]^{\mathrm{EL}}\langle i, j \rangle &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mu^{\mathrm{S}}(\mathrm{S}(\lambda l.\ \mathrm{nullDatum}\langle l[\mathrm{scriptPubKey}], \lceil j \rceil \rangle)) \\
&\qquad (e[\mathrm{tx}][\mathrm{outputs}]\lceil i \rceil)) \\
[\![\mathsf{outputSurjectionProof}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\ \mathrm{PrfHash}(l[\mathrm{asset}]))(e[\mathrm{tx}][\mathrm{outputs}]\lceil i \rceil)) \\
[\![\mathsf{outputRangeProof}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{S}(\lambda l.\ \mathrm{PrfHash}(l[\mathrm{amount}]))(e[\mathrm{tx}][\mathrm{outputs}]\lceil i \rceil)) \\
[\![\mathsf{totalFee}]\!]^{\mathrm{EL}}(a) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(\mathrm{fee}\langle e[\mathrm{tx}][\mathrm{outputs}], a \rangle) \\
[\![\mathsf{tapleafVersion}]\!]^{\mathrm{EL}}\langle\rangle &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{tapEnv}][\mathrm{leafVersion}]) \\
[\![\mathsf{tapbranch}]\!]^{\mathrm{EL}}(i) &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{tapEnv}][\mathrm{branch}]\lceil i \rceil) \\
[\![\mathsf{internalKey}]\!]^{\mathrm{EL}}\langle\rangle &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{tapEnv}][\mathrm{internalKey}]) \\
[\![\mathsf{genesisBlockHash}]\!]^{\mathrm{EL}}\langle\rangle &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{genesisBlockHash}]) \\
[\![\mathsf{scriptCMR}]\!]^{\mathrm{EL}}\langle\rangle &:= \lambda e : \mathrm{ELEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{scriptCMR}])
\end{aligned}
$$

where

$$
\begin{aligned}
\mathrm{PrfHash}(\sigma^{\mathbf{L}}(a)) &:= \mathrm{SHA256}(a[\mathrm{prf}]) \\
\mathrm{PrfHash}(\sigma^{\mathbf{R}}(a)) &:= \mathrm{SHA256}(\epsilon) \\
\mathrm{OptionPrfHash}(\eta^{S}(a)) &:= \mathrm{PrfHash}(a) \\
\mathrm{OptionPrfHash}(\emptyset^{\mathrm{S}}) &:= \mathrm{SHA256}(\epsilon) \\
\mathrm{ClearAsset}(a) &:= \mathrm{Asset}(\mathsf{unit})(a) \\
\mathrm{ClearAmount}(a) &:= \mathrm{Amount}(\mathsf{unit})(a) \\
\mathrm{isNewIssuance}(\eta^{S}(\sigma^{\mathbf{L}}(l))) &:= \eta^{S}(l) \\
\mathrm{isNewIssuance}(\eta^{S}(\sigma^{\mathbf{R}}(r))) &:= \emptyset^{\mathrm{S}} \\
\mathrm{isNewIssuance}(\emptyset^{\mathrm{S}}) &:= \emptyset^{\mathrm{S}} \\
\mathrm{isReissuance}(\eta^{S}(\sigma^{\mathbf{L}}(l))) &:= \emptyset^{\mathrm{S}} \\
\mathrm{isReissuance}(\eta^{S}(\sigma^{\mathbf{R}}(r))) &:= \eta^{S}(r) \\
\mathrm{isReissuance}(\emptyset^{\mathrm{S}}) &:= \emptyset^{\mathrm{S}} \\
\mathrm{newIssuance}(l) &:= \mathrm{isNewIssuance}(l[\mathrm{issuance}]) \\
\mathrm{reissuance}(l) &:= \mathrm{isReissuance}(l[\mathrm{issuance}]) \\
\mathrm{getIssuanceAssetAmt}(\sigma^{\mathbf{L}}(l)) &:= l[\mathrm{amount}] \\
\mathrm{getIssuanceAssetAmt}(\sigma^{\mathbf{R}}(r)) &:= r[\mathrm{amount}] \\
\mathrm{getIssuanceTokenAmt}(\sigma^{\mathbf{L}}(l)) &:= l[\mathrm{tokenAmount}] \\
\mathrm{getIssuanceTokenAmt}(\sigma^{\mathbf{R}}(r)) &:= \sigma^{\mathbf{R}}\lfloor 0 \rfloor_{64} \\
\mathrm{issuanceAssetAmt}(l) &:= \mathrm{S}(\mathrm{getIssuanceAssetAmt})(l[\mathrm{issuance}]) \\
\mathrm{issuanceTokenAmt}(l) &:= \mathrm{S}(\mathrm{getIssuanceTokenAmt})(l[\mathrm{issuance}]) \\
\mathrm{fee}\langle \epsilon, a \rangle &:= \lfloor 0 \rfloor_{64} \\
\mathrm{fee}\langle o \blacktriangleleft l, a \rangle &:= \mathrm{fee}\langle l, a \rangle \lfloor + \rfloor_{64} v && \text{when } o[\mathrm{asset}] = \sigma^{\mathbf{R}}(a) \\
& && \text{and } o[\mathrm{value}] = \sigma^{\mathbf{R}}(v) \\
& && \text{and } o[\mathrm{scriptPubKey}] = \epsilon \\
\mathrm{fee}\langle o \blacktriangleleft l, a \rangle &:= \mathrm{fee}\langle l, a \rangle && \text{when } o[\mathrm{asset}] \neq \sigma^{\mathbf{R}}(a) \\
& && \text{or } \forall v.\ o[\mathrm{value}] \neq \sigma^{\mathbf{R}}(v) \\
& && \text{or } o[\mathrm{scriptPubKey}] \neq \epsilon \\
\mathrm{encAsset}(\emptyset^{\mathrm{S}}) &:= \mathtt{[00]}_{2^8} \\
\mathrm{encAsset}(\eta^{\mathrm{S}}(\sigma^{\mathbf{R}}(a))) &:= \mathtt{[01]}_{2^8} \cdot \mathrm{BE}_{256}(a)
\end{aligned}
$$

$$
\begin{aligned}
\text{encAsset}(\eta^{\mathrm{S}}(\sigma^{\mathbf{L}}\langle \mathtt{0}_2, x\rangle)) &:= \mathtt{[0a]}_{2^8}\cdot\text{BE}_{256}(x[\text{pt}]) \\
\text{encAsset}(\eta^{\mathrm{S}}(\sigma^{\mathbf{L}}\langle \mathtt{1}_2, x\rangle)) &:= \mathtt{[0b]}_{2^8}\cdot\text{BE}_{256}(x[\text{pt}]) \\
\text{encAmt}(\sigma^{\mathbf{R}}\lfloor 0\rfloor_{64}) &:= \mathtt{[00]}_{2^8} \\
\text{encAmt}(\sigma^{\mathbf{R}}(v)) &:= \mathtt{[01]}_{2^8}\cdot\text{BE}_{64}(v) && \text{when } v \neq \lfloor 0\rfloor_{64} \\
\text{encAmt}(\sigma^{\mathbf{L}}\langle \mathtt{0}_2, x\rangle) &:= \mathtt{[08]}_{2^8}\cdot\text{BE}_{256}(x[\text{pt}]) \\
\text{encAmt}(\sigma^{\mathbf{L}}\langle \mathtt{1}_2, x\rangle) &:= \mathtt{[09]}_{2^8}\cdot\text{BE}_{256}(x[\text{pt}]) \\
\text{encNonce}(\emptyset^{\mathrm{S}}) &:= \mathtt{[00]}_{2^8} \\
\text{encNonce}(\eta^{\mathrm{S}}(\sigma^{\mathbf{R}}(a))) &:= \mathtt{[01]}_{2^8}\cdot\text{BE}_{256}(a) \\
\text{encNonce}(\eta^{\mathrm{S}}(\sigma^{\mathbf{L}}\langle \mathtt{0}_2, x\rangle)) &:= \mathtt{[02]}_{2^8}\cdot\text{BE}_{256}(x[\text{pt}]) \\
\text{encNonce}(\eta^{\mathrm{S}}(\sigma^{\mathbf{L}}\langle \mathtt{1}_2, x\rangle)) &:= \mathtt{[03]}_{2^8}\cdot\text{BE}_{256}(x[\text{pt}]) \\
\text{encProof}(\emptyset^{\mathrm{S}}) &:= \text{BE}_{256}(\text{SHA256}(\epsilon)) \\
\text{encProof}(\eta^{\mathrm{S}}(h)) &:= \text{BE}_{256}(h) \\
\text{encIssuance}(\emptyset^{\mathrm{S}}) &:= \mathtt{[00]}_{2^8}\cdot\mathtt{[00]}_{2^8} \\
\text{encIssuance}(\eta^{\mathrm{S}}(\sigma^{\mathbf{L}}(x))) &:= \text{encAmt}(x[\text{amount}])\cdot \\
&\quad \text{encAmt}(x[\text{tokenAmount}])\cdot \\
&\quad \text{BE}_{256}(\lfloor 0\rfloor_{256})\cdot \\
&\quad \text{BE}_{256}(x[\text{contractHash}])\cdot \\
&\quad \text{encProof}(\text{AmountPrfHash}(x[\text{amount}]))\cdot \\
&\quad \text{encProof}(\text{AmountPrfHash}(x[\text{tokenAmount}])) \\
\text{encIssuance}(\eta^{\mathrm{S}}(\sigma^{\mathbf{R}}(x))) &:= \text{encAmt}(x[\text{amount}])\cdot \\
&\quad \mathtt{[00]}_{2^8}\cdot \\
&\quad \text{BE}_{256}(x[\text{blindingNonce}])\cdot \\
&\quad \text{BE}_{256}(x[\text{entropy}])\cdot \\
&\quad \text{encProof}(\text{AmountPrfHash}(x[\text{amount}]))\cdot \\
&\quad \text{encProof}(\emptyset^{\mathrm{S}})
\end{aligned}
$$

where nullDatum is defined in the next section.

A.1.1 Null Data

An output's scriptPubKey is call a `TX_NULL_DATA` script if it consists of an `OP_RETURN` opcode ($\lceil \mathtt{6a}_{2^8} \rceil$) followed by "`PushOnly`", operations, which are operations whose opcode is `OP_16` (96) or less (including `OP_RESERVED` (80)).

In Elements, pegouts have an instance of a `TX_NULL_DATA` scriptPubKey. For that reason we offer primitives for detecting and parsing `TX_NULL_DATA` scriptPubKeys. Below we define the function

$$\text{nullData} : (2^8)^* \to \text{S}((2^2 \times 2^{256} + (2 + 2^4))^*)$$

which,

1. decides if a script is "`PushOnly`", and
2. if it is push only, returns a list of parsed opcodes in which the pushed data is hashed.

We also define

$$\text{nullData} : (2^8)^* \times \mathbb{N} \to \text{S}(\text{S}(2^2 \times 2^{256} + (2 + 2^4)))$$

which decides if a script is `TX_NULL_DATA` and returns the parsed element at the given index for the parsed data, if length of the parsed script is long enough. This is used to define the semantics of outputNullDatum above.

$$
\begin{aligned}
\mathrm{nullData}(\epsilon_{2^8}) &:= \eta^{\mathrm{S}}(\epsilon_{2^2\times 2^{256}+(2+2^4)}) \\
\mathrm{nullData}(n \blacktriangleleft d{\cdot}l) &:= \mu^{\mathrm{S}}(\mathrm{S}(\lambda r.\sigma^{\mathbf{L}}\langle\lfloor 0\rfloor_2, \mathrm{SHA256}(d)\rangle \blacktriangleleft r)(\mathrm{nullData}(l))) \\
&\qquad \text{when } |d| = \lceil n\rceil_8 < 76 \\
\mathrm{nullData}(n \blacktriangleleft l) &:= \emptyset^{\mathrm{S}} \qquad \text{when } |l| < \lceil n\rceil_8 < 76 \\
\mathrm{nullData}(\lfloor 76\rfloor_8 \blacktriangleleft n \blacktriangleleft d{\cdot}l) &:= \mu^{\mathrm{S}}(\mathrm{S}(\lambda r.\sigma^{\mathbf{L}}\langle\lfloor 1\rfloor_2, \mathrm{SHA256}(d)\rangle \blacktriangleleft r)(\mathrm{nullData}(l))) \\
&\qquad \text{when } |d| = \lceil n\rceil_8 \\
\mathrm{nullData}(\lfloor 76\rfloor_8 \blacktriangleleft n \blacktriangleleft l) &:= \emptyset^{\mathrm{S}} \qquad \text{when } |l| < \lceil n\rceil_8 \\
\mathrm{nullData}(\lfloor 76\rfloor_8 \blacktriangleleft \varepsilon) &:= \emptyset^{\mathrm{S}} \\
\mathrm{nullData}(\lfloor 77\rfloor_8 \blacktriangleleft n_1 \blacktriangleleft n_0 \blacktriangleleft d{\cdot}l) &:= \mu^{\mathrm{S}}(\mathrm{S}(\lambda r.\sigma^{\mathbf{L}}\langle\lfloor 2\rfloor_2, \mathrm{SHA256}(d)\rangle \blacktriangleleft r)(\mathrm{nullData}(l))) \\
&\qquad \text{when } |d| = \lceil\langle n_0, n_1\rangle\rceil_{16} \\
\mathrm{nullData}(\lfloor 77\rfloor_8 \blacktriangleleft n_1 \blacktriangleleft n_0 \blacktriangleleft l) &:= \emptyset^{\mathrm{S}} \qquad \text{when } |l| < \lceil\langle n_0, n_1\rangle\rceil_{16} \\
\mathrm{nullData}(\lfloor 77\rfloor_8 \blacktriangleleft l) &:= \emptyset^{\mathrm{S}} \qquad \text{when } |l| < 2 \\
\mathrm{nullData}(\lfloor 78\rfloor_8 \blacktriangleleft n_3 \blacktriangleleft n_2 \blacktriangleleft n_1 \blacktriangleleft n_0 \blacktriangleleft d{\cdot}l) &:= \mu^{\mathrm{S}}(\mathrm{S}(\lambda r.\sigma^{\mathbf{L}}\langle\lfloor 3\rfloor_2, \mathrm{SHA256}(d)\rangle \blacktriangleleft r)(\mathrm{nullData}(l))) \\
&\qquad \text{when } |d| = \lceil\langle\langle n_0, n_1\rangle, \langle n_2, n_3\rangle\rangle\rceil_{32} \\
\mathrm{nullData}(\lfloor 78\rfloor_8 \blacktriangleleft n_3 \blacktriangleleft n_2 \blacktriangleleft n_1 \blacktriangleleft n_0 \blacktriangleleft l) &:= \emptyset^{\mathrm{S}} \qquad \text{when } |l| < \lceil\langle\langle n_0, n_1\rangle, \langle n_2, n_3\rangle\rangle\rceil_{32} \\
\mathrm{nullData}(\lfloor 78\rfloor_8 \blacktriangleleft l) &:= \emptyset^{\mathrm{S}} \qquad \text{when } |l| < 4 \\
\mathrm{nullData}(\lfloor 79\rfloor_8 \blacktriangleleft l) &:= \mu^{\mathrm{S}}(\mathrm{S}(\lambda r.\sigma^{\mathbf{R}}(\sigma^{\mathbf{L}}(0_2)) \blacktriangleleft r)(\mathrm{nullData}(l))) \\
\mathrm{nullData}(\lfloor 80\rfloor_8 \blacktriangleleft l) &:= \mu^{\mathrm{S}}(\mathrm{S}(\lambda r.\sigma^{\mathbf{R}}(\sigma^{\mathbf{L}}(1_2)) \blacktriangleleft r)(\mathrm{nullData}(l))) \\
\mathrm{nullData}(x \blacktriangleleft l) &:= \mu^{\mathrm{S}}(\mathrm{S}(\lambda r.\sigma^{\mathbf{R}}(\sigma^{\mathbf{R}}\lfloor\lceil x\rceil_8 - 81\rfloor_4) \blacktriangleleft r)(\mathrm{nullData}(l))) \\
&\qquad \text{when } 81 \le \lceil x\rceil_8 < 97 \\
\mathrm{nullData}(x \blacktriangleleft l) &:= \emptyset^{\mathrm{S}} \qquad \text{when } 97 \le \lceil x\rceil_8 \\
\mathrm{nullDatum}\langle \mathsf{6a}_{2^8} \blacktriangleleft s, j\rangle &:= \mathrm{S}(\lambda l.l[j])(\mathrm{nullData}(s)) \\
\mathrm{nullDatum}\langle x \blacktriangleleft s, j\rangle &:= \emptyset^{\mathrm{S}} \qquad \text{when } x \neq \mathsf{6a}_{2^8} \\
\mathrm{nullDatum}\langle \epsilon, j\rangle &:= \emptyset^{\mathrm{S}}
\end{aligned}
$$

A.1.2 Merkle Roots

We extend the definition of the commitment Merkle root to support the new expressions by hashing new unique byte strings.

$$
\begin{aligned}
\#^c(\mathsf{version}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[766572736 96f6e]}}_{\mathrm{IV}} \\
\#^c(\mathsf{lockTime}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[6c6f636b54696d65]}}_{\mathrm{IV}} \\
\#^c(\mathsf{inputsHash}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[696e7075747348617368]}}_{\mathrm{IV}} \\
\#^c(\mathsf{outputsHash}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[6f75747075747348617368]}}_{\mathrm{IV}} \\
\#^c(\mathsf{inputPegin}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[696e707574506567696e]}}_{\mathrm{IV}} \\
\#^c(\mathsf{inputPrevOutpoint}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[696e70757450726 4f7574706f696e74]}}_{\mathrm{IV}} \\
\#^c(\mathsf{inputAsset}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[696e7075744173736574]}}_{\mathrm{IV}} \\
\#^c(\mathsf{inputAmount}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[696e707574416d6f756e74]}}_{\mathrm{IV}} \\
\#^c(\mathsf{inputScriptHash}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[696e70757453637269707448617368]}}_{\mathrm{IV}} \\
\#^c(\mathsf{inputSequence}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[696e7075745365 7175656e6365]}}_{\mathrm{IV}} \\
\#^c(\mathsf{inputAnnexHash}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[696e707574416e6e657848617368]}}_{\mathrm{IV}} \\
\#^c(\mathsf{inputScriptSigHash}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[696e70757453637269707453696748617368]}}_{\mathrm{IV}} \\
\#^c(\mathsf{reissuanceBlinding}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[726569737375616e6365426c696e64696e67]}}_{\mathrm{IV}}
\end{aligned}
$$

$$
\begin{aligned}
\#^c(\mathsf{newIssuanceContract}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[6e657749737375616e6365436f6e7472616374]}}_{\mathrm{IV}} \\
\#^c(\mathsf{reissuanceEntropy}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[726569737375616e6365456e74726f7079]}}_{\mathrm{IV}} \\
\#^c(\mathsf{issuanceAssetAmount}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[69737375616e63654173736574416d6f756e74]}}_{\mathrm{IV}} \\
\#^c(\mathsf{issuanceTokenAmount}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[69737375616e6365546f6b656e416d6f756e74]}}_{\mathrm{IV}} \\
\#^c(\mathsf{issuanceAssetProof}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[69737375616e6365417373657450726f6f66]}}_{\mathrm{IV}} \\
\#^c(\mathsf{issuanceTokenProof}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[69737375616e6365546f6b656e50726f6f66]}}_{\mathrm{IV}} \\
\#^c(\mathsf{currentIndex}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[63757272656e74496e646578]}}_{\mathrm{IV}} \\
\#^c(\mathsf{outputAsset}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[6f7574707574173736574]}}_{\mathrm{IV}} \\
\#^c(\mathsf{outputAmount}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[6f7574707574416d6f756e74]}}_{\mathrm{IV}} \\
\#^c(\mathsf{outputNonce}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[6f75747075744e6f6e6365]}}_{\mathrm{IV}} \\
\#^c(\mathsf{outputScriptHash}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[6f75747075745363726970744861736 8]}}_{\mathrm{IV}} \\
\#^c(\mathsf{outputNullDatum}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[6f75747075744e756c6c446174756d]}}_{\mathrm{IV}} \\
\#^c(\mathsf{outputSurjectionProof}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[6f7574707574537572 6a656374696f6e50726f6f66]}}_{\mathrm{IV}} \\
\#^c(\mathsf{outputRangeProof}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[6f757470757452616e676550726f6f66]}}_{\mathrm{IV}} \\
\#^c(\mathsf{fee}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[666565]}}_{\mathrm{IV}} \\
\#^c(\mathsf{tapleafVersion}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[7461706c65616565657273696f6e]}}_{\mathrm{IV}} \\
\#^c(\mathsf{tapbranch}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[7461706272616e6368]}}_{\mathrm{IV}} \\
\#^c(\mathsf{internalKey}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[696e7465726e616c4b6579]}}_{\mathrm{IV}} \\
\#^c(\mathsf{genesisBlockHash}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[67656e65736973426c6f636b48617368]}}_{\mathrm{IV}} \\
\#^c(\mathsf{scriptCMR}) &:= \mathrm{SHA256}^{\mathrm{ELprefix}\cdot\texttt{[736372697074434d52]}}_{\mathrm{IV}}
\end{aligned}
$$

where

$\mathrm{ELprefix} := \texttt{[53696d706c69636974792d44726166741f5072696d69746976651f456c656d6e74731f]}$

Appendix B
Catelogue of Jets

UNDER DEVELOPMENT AND SUBJECT TO CHANGE

We develop a recommended set of jets and provide an interm encoding. An encoding ought to be based on how frequenly jets are used, however we do not currently have good estimates of that. As an interm measure we develop a heirarchical encoding of jets by category.

The properties for jets listed below may not fully define the jet's semantics. All jets will be formally specified by a Simplicity program that implements their complete semantics. Those formal specificaitons will be found in the Coq library. Implementations MUST implement the COMPLETE specifications as defined in the Coq library.

B.1 `110...`: Core Jets

The following jets are specified in core Simplicity or Simplicity with assertions, and therefore are applicable to any Simplicity application.

B.1.1 `1100...`: Jets for multi-bit logic

It is recomended that jets be supported for multi-bit words up to 2^{256} in size.

TODO: define $\mathsf{bit}_{n,m} : 2^n \to 2$

B.1.1.1 verify

$\ulcorner \text{'verify'} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 1 \urcorner$
$\mathsf{verify} : 2 \vdash \mathbb{1}$

B.1.1.2 low

$\ulcorner \text{'low'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 2 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{low}_{2^n} : \mathbb{1} \vdash 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil \mathsf{bit}_{2^n,i}(\llbracket \mathsf{low}_{2^n} \rrbracket \langle\rangle) \rceil_1 = 0$
$\lceil \llbracket \mathsf{low}_{2^n} \rrbracket \langle\rangle \rceil_{2^n} = 0$

Aliases:
$\mathsf{zero}_{2^n} := \mathsf{low}_{2^n} : \mathbb{1} \vdash 2^{2^n}$

B.1.1.3 high

$\ulcorner \text{'high'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 3 \urcorner \cdot \ulcorner n+1 \urcorner$

$\mathsf{high}_{2^n}{:}\mathbb{1} \vdash 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil \mathrm{bit}_{2^n,i}(\llbracket \mathsf{high}_{2^n} \rrbracket \langle\rangle) \rceil_1 = 1$
$\lceil \llbracket \mathsf{high}_{2^n} \rrbracket \langle\rangle \rceil_{2^n} = 2^n - 1$

Aliases:
$\mathsf{one}_1 := \mathsf{high}_1{:}\mathbb{1} \vdash 2$

B.1.1.4 complement

$\ulcorner \text{'}\mathsf{complement}\text{'}_{2^n} \urcorner := [\mathtt{110}]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 4 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{complement}_{2^n} : 2^{2^n} \vdash 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil \mathrm{bit}_{2^n,i}(\llbracket \mathsf{complement}_{2^n} \rrbracket(x)) \rceil_1 = 1 - \lceil \mathrm{bit}_{2^n,i}(x) \rceil_1$
$\lceil \llbracket \mathsf{complement}_{2^n} \rrbracket(x) \rceil_{2^n} = 2^n - 1 - \lceil x \rceil_{2^n}$

B.1.1.5 and

$\ulcorner \text{'}\mathsf{and}\text{'}_{2^n} \urcorner := [\mathtt{110}]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 5 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{and}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil \mathrm{bit}_{2^n,i}(\llbracket \mathsf{and}_{2^n} \rrbracket \langle x, y \rangle) \rceil_1 = \chi(\lceil \mathrm{bit}_{2^n,i}(x) \rceil_1 = 1 \wedge \lceil \mathrm{bit}_{2^n,i}(y) \rceil_1 = 1)$

B.1.1.6 or

$\ulcorner \text{'}\mathsf{or}\text{'}_{2^n} \urcorner := [\mathtt{110}]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 6 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{or}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil \mathrm{bit}_{2^n,i}(\llbracket \mathsf{or}_{2^n} \rrbracket \langle x, y \rangle) \rceil_1 = \chi(\lceil \mathrm{bit}_{2^n,i}(x) \rceil_1 = 1 \vee \lceil \mathrm{bit}_{2^n,i}(y) \rceil_1 = 1)$

B.1.1.7 xor

$\ulcorner \text{'}\mathsf{xor}\text{'}_{2^n} \urcorner := [\mathtt{110}]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 7 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{xor}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil \mathrm{bit}_{2^n,i}(\llbracket \mathsf{xor}_{2^n} \rrbracket \langle x, y \rangle) \rceil_1 \equiv \lceil \mathrm{bit}_{2^n,i}(x) \rceil_1 + \lceil \mathrm{bit}_{2^n,i}(y) \rceil_1 (\mathrm{mod}\, 2)$

B.1.1.8 maj

$\ulcorner \text{'}\mathsf{maj}\text{'}_{2^n} \urcorner := [\mathtt{110}]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 8 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{maj}_{2^n} : 2^{2^n} \times (2^{2^n} \times 2^{2^n}) \vdash 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil \mathrm{bit}_{2^n,i}(\llbracket \mathsf{maj}_{2^n} \rrbracket \langle x, \langle y, z \rangle \rangle) \rceil_1 = \chi(2 \leq \lceil \mathrm{bit}_{2^n,i}(x) \rceil_1 + \lceil \mathrm{bit}_{2^n,i}(y) \rceil_1 + \lceil \mathrm{bit}_{2^n,i}(z) \rceil_1)$

B.1.1.9 xor-xor

$\ulcorner$'xor-xor'$_{2^n}\urcorner := [110]_2.\ulcorner 1\urcorner.\ulcorner 9\urcorner.\ulcorner n+1\urcorner$
$\mathsf{xor\text{-}xor}_{2^n} : 2^{2^n} \times (2^{2^n} \times 2^{2^n}) \vdash 2^{2^n}$ for $0 \le n$

Properties:
$\lceil \mathrm{bit}_{2^n,i}([\![\mathsf{xor\text{-}xor}_{2^n}]\!]\langle x, \langle y, z\rangle\rangle)\rceil_1 \equiv \lceil \mathrm{bit}_{2^n,i}(x)\rceil_1 + \lceil \mathrm{bit}_{2^n,i}(y)\rceil_1 + \lceil \mathrm{bit}_{2^n,i}(z)\rceil_1 (\mathrm{mod}\, 2)$

B.1.1.10 ch

$\ulcorner$'ch'$_{2^n}\urcorner := [110]_2.\ulcorner 1\urcorner.\ulcorner 10\urcorner.\ulcorner n+1\urcorner$
$\mathsf{ch}_{2^n} : 2^{2^n} \times (2^{2^n} \times 2^{2^n}) \vdash 2^{2^n}$ for $0 \le n$

Properties:
$\lceil \mathrm{bit}_{2^n,i}([\![\mathsf{ch}_{2^n}]\!]\langle x, \langle y, z\rangle\rangle)\rceil_1 = (1 - \lceil \mathrm{bit}_{2^n,i}(x)\rceil_1) \cdot \lceil \mathrm{bit}_{2^n,i}(y)\rceil_1 + \lceil \mathrm{bit}_{2^n,i}(x)\rceil_1 \cdot \lceil \mathrm{bit}_{2^n,i}(z)\rceil_1$

B.1.1.11 some

(CAUTION: Not defined when $n = 0$.)
$\ulcorner$'some'$_{2^n}\urcorner := [110]_2.\ulcorner 1\urcorner.\ulcorner 11\urcorner.\ulcorner n\urcorner$
$\mathsf{some}_{2^n} : 2^{2^n} \vdash 2$ for $1 \le n$

Properties:
$\lceil [\![\mathsf{some}_{2^n}]\!](x)\rceil_1 = \chi(\lceil x\rceil_{2^n} \neq 0)$

Aliases:
$\mathsf{some}_1 := \mathsf{iden} : 2 \vdash 2$

B.1.1.12 all

(CAUTION: Not defined when $n = 0$.)
$\ulcorner$'all'$_{2^n}\urcorner := [110]_2.\ulcorner 1\urcorner.\ulcorner 12\urcorner.\ulcorner n\urcorner$
$\mathsf{all}_{2^n} : 2^{2^n} \vdash 2$ for $1 \le n$

Properties:
$\lceil [\![\mathsf{all}_{2^n}]\!](x)\rceil_1 = \chi(\lceil x\rceil_{2^n} = 2^n - 1)$

Aliases:
$\mathsf{all}_1 := \mathsf{iden} : 2 \vdash 2$

B.1.1.13 eq

$\ulcorner$'eq'$_{2^n}\urcorner := [110]_2.\ulcorner 1\urcorner.\ulcorner 13\urcorner.\ulcorner n+1\urcorner$
$\mathsf{eq}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2$ for $0 \le n$

Properties:
$[\![\mathsf{eq}_{2^n}]\!]\langle x, y\rangle = \chi(x = y)$

B.1.1.14 full-left-shift

$\ulcorner$'full-left-shift'$_{2^n,2^m}\urcorner := [110]_2.\ulcorner 1\urcorner.\ulcorner 14\urcorner.\ulcorner m+1\urcorner.\ulcorner n-m\urcorner$
$\mathsf{full\text{-}left\text{-}shift}_{2^n,2^m} : 2^{2^n} \times 2^{2^m} \vdash 2^{2^m} \times 2^{2^n}$ for $0 \le m < n$

Properties:
$\mathrm{bit}_{2^m,i}(\pi_1([\![\mathsf{full\text{-}left\text{-}shift}_{2^n,2^m}]\!]\langle x, y\rangle)) = \mathrm{bit}_{2^n,i}(x)$ for $0 \le i < 2^m$
$\mathrm{bit}_{2^n,i}(\pi_2([\![\mathsf{full\text{-}left\text{-}shift}_{2^n,2^m}]\!]\langle x, y\rangle)) = \mathrm{bit}_{2^n,2^m+i}(x)$ for $0 \le i < 2^n - 2^m$
$\mathrm{bit}_{2^n,2^n-2^m+i}(\pi_2([\![\mathsf{full\text{-}left\text{-}shift}_{2^n,2^m}]\!]\langle x, y\rangle)) = \mathrm{bit}_{2^m,i}(y)$ for $0 \le i < 2^m$

Aliases:
$\mathsf{full\text{-}left\text{-}shift}_{2^n,2^n} := \mathsf{iden} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n} \times 2^{2^n}$
$\mathsf{full\text{-}left\text{-}shift}_{2^n,2^m} := \mathsf{full\text{-}right\text{-}shift}_{2^m,2^n} : 2^{2^n} \times 2^{2^m} \vdash 2^{2^m} \times 2^{2^n}$ when $0 \leq n < m$
$\mathsf{full\text{-}right\text{-}shift}_{2^n,2^m} := \mathsf{full\text{-}left\text{-}shift}_{2^m,2^n} : 2^{2^m} \times 2^{2^n} \vdash 2^{2^n} \times 2^{2^m}$ when $0 \leq n < m$

B.1.1.15 full-right-shift

$\ulcorner \mathsf{'full\text{-}right\text{-}shift'}_{2^n,2^m} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 15 \urcorner \cdot \ulcorner m+1 \urcorner \cdot \ulcorner n-m \urcorner$
$\mathsf{full\text{-}right\text{-}shift}_{2^n,2^m} : 2^{2^m} \times 2^{2^n} \vdash 2^{2^n} \times 2^{2^m}$ for $0 \leq m < n$

Properties:
$\mathrm{bit}_{2^n,i}(\pi_1([\![\mathsf{full\text{-}right\text{-}shift}_{2^n,2^m}]\!]\langle x, y\rangle)) = \mathrm{bit}_k(x)$ for $0 \leq i < 2^m$
$\mathrm{bit}_{2^n,2^m+i}(\pi_1([\![\mathsf{full\text{-}right\text{-}shift}_{2^n,2^m}]\!]\langle x, y\rangle)) = \mathrm{bit}_{2^n,i}(y)$ for $0 \leq i < 2^n - 2^m$
$\mathrm{bit}_{2^m,i}(\pi_2([\![\mathsf{full\text{-}right\text{-}shift}_{2^n,2^m}]\!]\langle x, y\rangle)) = \mathrm{bit}_{2^n,2^n-2^m+i}(y)$ for $0 \leq i < 2^m$

Aliases:
$\mathsf{full\text{-}right\text{-}shift}_{2^n,2^n} := \mathsf{iden} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n} \times 2^{2^n}$
$\mathsf{full\text{-}right\text{-}shift}_{2^n,2^m} := \mathsf{full\text{-}left\text{-}shift}_{2^m,2^n} : 2^{2^n} \times 2^{2^m} \vdash 2^{2^m} \times 2^{2^n}$ when $0 \leq n < m$
$\mathsf{full\text{-}left\text{-}shift}_{2^n,2^m} := \mathsf{full\text{-}right\text{-}shift}_{2^m,2^n} : 2^{2^n} \times 2^{2^m} \vdash 2^{2^m} \times 2^{2^n}$ when $0 \leq n < m$

B.1.1.16 leftmost

$\ulcorner \mathsf{'leftmost'}_{2^n,2^m} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 16 \urcorner \cdot \ulcorner m+1 \urcorner \cdot \ulcorner n-m \urcorner$
$\mathsf{leftmost}_{2^n,2^m} : 2^{2^n} \vdash 2^{2^m}$ for $0 \leq m < n$

Properties:
$\mathrm{bit}_{2^m,i}([\![\mathsf{leftmost}_{2^n,2^m}]\!](x)) = \mathrm{bit}_{2^n,i}(x)$

Aliases:
$\mathsf{leftmost}_{2^n,2^n} := \mathsf{iden} : 2^{2^n} \vdash 2^{2^n}$

B.1.1.17 rightmost

$\ulcorner \mathsf{'rightmost'}_{2^n,2^m} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 17 \urcorner \cdot \ulcorner m+1 \urcorner \cdot \ulcorner n-m \urcorner$
$\mathsf{rightmost}_{2^n,2^m} : 2^{2^n} \vdash 2^{2^m}$ for $0 \leq m < n$

Properties:
$\mathrm{bit}_{2^m,i}([\![\mathsf{rightmost}_{2^n,2^m}]\!](x)) = \mathrm{bit}_{2^n,2^n-2^m+i}(x)$
$\lceil [\![\mathsf{rightmost}_{2^n,2^m}]\!](x) \rceil_{2^m} = \lceil x \rceil_{2^n} \pmod{2^m}$

Aliases:
$\mathsf{rightmost}_{2^n,2^n} := \mathsf{iden} : 2^{2^n} \vdash 2^{2^n}$

B.1.1.18 left-pad-low

(Note: derived from full-left-shift.)
$\ulcorner \mathsf{'left\text{-}pad\text{-}low'}_{2^n,2^m} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 18 \urcorner \cdot \ulcorner n+1 \urcorner \cdot \ulcorner m-n \urcorner$
$\mathsf{left\text{-}pad\text{-}low}_{2^n,2^m} : 2^{2^n} \vdash 2^{2^m}$ for $0 \leq n < m$

Properties:
$\lceil \mathrm{bit}_{2^m,i}([\![\mathsf{left\text{-}pad\text{-}low}_{2^n,2^m}]\!](x)) \rceil_1 = 0$ when $i < 2^m - 2^n$
$\mathrm{bit}_{2^m,2^m-2^n+i}([\![\mathsf{left\text{-}pad\text{-}low}_{2^n,2^m}]\!](x)) = \mathrm{bit}_{2^n,i}(x)$ when $i < 2^n$

$\lceil [\![\mathsf{left\text{-}pad\text{-}low}_{2^n,2^m}]\!](x) \rceil_{2^m} = \lceil x \rceil_{2^n}$

B.1.1.19 left-pad-high

(Note: derived from full-left-shift.)
$\ulcorner$'left-pad-high'$_{2^n,2^m}\urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 19 \urcorner \cdot \ulcorner n+1 \urcorner \cdot \ulcorner m-n \urcorner$
$\mathsf{left\text{-}pad\text{-}high}_{2^n,2^m} : 2^{2^n} \vdash 2^{2^m}$ for $0 \le n < m$

Properties:
$\lceil \mathrm{bit}_{2^m,i}([\![\mathsf{left\text{-}pad\text{-}high}_{2^n,2^m}]\!](x)) \rceil_1 = 1$ when $i < 2^m - 2^n$
$\mathrm{bit}_{2^m,2^m-2^n+i}([\![\mathsf{left\text{-}pad\text{-}high}_{2^n,2^m}]\!](x)) = \mathrm{bit}_{2^n,i}(x)$ when $i < 2^n$

B.1.1.20 left-extend

(Note: derived from leftmost, left-pad-low and left-pad-high.)
$\ulcorner$'left-extend'$_{2^n,2^m}\urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 20 \urcorner \cdot \ulcorner n+1 \urcorner \cdot \ulcorner m-n \urcorner$
$\mathsf{left\text{-}extend}_{2^n,2^m} : 2^{2^n} \vdash 2^{2^m}$ for $0 \le n < m$

Properties:
$\mathrm{bit}_{2^m,i}([\![\mathsf{left\text{-}extend}_{2^n,2^m}]\!](x)) = \mathrm{bit}_{2^n,0}(x)$ when $i < 2^n - 2$
$\mathrm{bit}_{2^m,2^m-2^n+i}([\![\mathsf{left\text{-}extend}_{2^n,2^m}]\!](x)) = \mathrm{bit}_{2^n,i}(x)$ when $i < 2^n$

Aliases:
$\mathsf{right\text{-}extend}_{1,2^m} := \mathsf{left\text{-}extend}_{1,2^m} : 2 \vdash 2^{2^m}$ for $0 \le n < m$

B.1.1.21 right-pad-low

(Note: derived from full-right-shift.)
$\ulcorner$'right-pad-low'$_{2^n,2^m}\urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 21 \urcorner \cdot \ulcorner n+1 \urcorner \cdot \ulcorner m-n \urcorner$
$\mathsf{right\text{-}pad\text{-}low}_{2^n,2^m} : 2^{2^n} \vdash 2^{2^m}$ for $0 \le n < m$

Properties:
$\mathrm{bit}_{2^m,i}([\![\mathsf{right\text{-}pad\text{-}low}_{2^n,2^m}]\!](x)) = \mathrm{bit}_{2^n,i}(x)$ when $i < 2^n$
$\lceil \mathrm{bit}_{2^m,i}([\![\mathsf{right\text{-}pad\text{-}low}_{2^n,2^m}]\!](x)) \rceil_1 = 0$ when $2^n \le i < 2^m$

B.1.1.22 right-pad-high

(Note: derived from full-right-shift.)
$\ulcorner$'right-pad-high'$_{2^n,2^m}\urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 22 \urcorner \cdot \ulcorner n+1 \urcorner \cdot \ulcorner m-n \urcorner$
$\mathsf{right\text{-}pad\text{-}high}_{2^n,2^m} : 2^{2^n} \vdash 2^{2^m}$ for $0 \le n < m$

Properties:
$\mathrm{bit}_{2^m,i}([\![\mathsf{right\text{-}pad\text{-}high}_{2^n,2^m}]\!](x)) = \mathrm{bit}_{2^n,i}(x)$ when $i < 2^n$
$\lceil \mathrm{bit}_{2^m,i}([\![\mathsf{right\text{-}pad\text{-}high}_{2^n,2^m}]\!](x)) \rceil_1 = 1$ when $2^n \le i < 2^m$

B.1.1.23 right-extend

(Note: derived from rightmost, right-pad-low, and right-pad-high.)
(CAUTION: Not defined when $n = 0$.)
$\ulcorner$'right-extend'$_{2^n,2^m}\urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 23 \urcorner \cdot \ulcorner n \urcorner \cdot \ulcorner m-n \urcorner$
$\mathsf{right\text{-}extend}_{2^n,2^m} : 2^{2^n} \vdash 2^{2^m}$ for $1 \le n < m$

Properties:
$\mathrm{bit}_{2^m,i}([\![\mathsf{right\text{-}extend}_{2^n,2^m}]\!](x)) = \mathrm{bit}_{2^n,i}(x)$ when $i < 2^n$
$\lceil \mathrm{bit}_{2^m,i}([\![\mathsf{right\text{-}extend}_{2^n,2^m}]\!](x)) \rceil_1 = \mathrm{bit}_{2^n,2^n-1}(x)$ when $2^n \le i < 2^m$

Aliases:

$\mathsf{right\text{-}extend}_{1,2^m} := \mathsf{left\text{-}extend}_{1,2^m} : 2 \vdash 2^{2^m}$ for $0 \leq n < m$

B.1.1.24 left-shift-with

Left shift by a given amount. Bits are filled with the provided value.

$\ulcorner \mathsf{'left\text{-}shift\text{-}with'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 24 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{left\text{-}shift\text{-}with}_{2^n} : 2 \times (2^{2^{\lceil \mathrm{lb}(n) \rceil}} \times 2^{2^n}) \vdash 2^{2^n}$ for $0 \leq n$

B.1.1.25 right-shift-with

Right shift by a given amount. Bits are filled with the provided value.
$\ulcorner \mathsf{'right\text{-}shift\text{-}with'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 25 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{right\text{-}shift\text{-}with}_{2^n} : 2 \times (2^{2^{\lceil \mathrm{lb}(n) \rceil}} \times 2^{2^n}) \vdash 2^{2^n}$ for $0 \leq n$

B.1.1.26 left-shift

Left shift by a given amount. Bits are filled with 0 bits.
$\ulcorner \mathsf{'left\text{-}shift'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 26 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{left\text{-}shift}_{2^n} : 2^{2^{\lceil \mathrm{lb}(n) \rceil}} \times 2^{2^n} \vdash 2^{2^n}$ for $0 \leq n$

B.1.1.27 right-shift

Right shift by a given amount. Bits are filled with 0 bits.
$\ulcorner \mathsf{'right\text{-}shift'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 27 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{right\text{-}shift}_{2^n} : 2^{2^{\lceil \mathrm{lb}(n) \rceil}} \times 2^{2^n} \vdash 2^{2^n}$ for $0 \leq n$

B.1.1.28 left-rotate

Right rotate by an amount.
$\ulcorner \mathsf{'left\text{-}rotate'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 28 \urcorner \cdot \ulcorner n \urcorner$
$\mathsf{left\text{-}rotate}_{2^n} : 2^{2^{\lceil \mathrm{lb}(n) \rceil}} \times 2^{2^n} \vdash 2^{2^n}$ for $1 \leq n$

Aliases:
$\mathsf{left\text{-}rotate}_1 := \mathsf{drop} : \mathbb{1} \times 2 \vdash 2$

B.1.1.29 right-rotate

Right rotate by an amount.
$\ulcorner \mathsf{'right\text{-}rotate'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 29 \urcorner \cdot \ulcorner n \urcorner$
$\mathsf{right\text{-}rotate}_{2^n} : 2^{2^{\lceil \mathrm{lb}(n) \rceil}} \times 2^{2^n} \vdash 2^{2^n}$ for $1 \leq n$

Aliases:
$\mathsf{right\text{-}rotate}_1 := \mathsf{drop} : \mathbb{1} \times 2 \vdash 2$

B.1.1.30 transpose

(CAUTION: Not defined when $n = 0$ or $m = 0$.)
(Note: Support only recommened up to $2^n \cdot 2^m \leq 256$.)
$\ulcorner \mathsf{'transpose'}_{2^n,2^m} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 30 \urcorner \cdot \ulcorner n \urcorner \cdot \ulcorner m \urcorner$
$\mathsf{transpose}_{2^n,2^m} : 2^{2^n \cdot 2^m} \vdash 2^{2^m \cdot 2^n}$ for $1 \leq n$ and $1 \leq m$

Aliases:
$\mathsf{transpose}_{1,2^n} := \mathsf{iden} : 2^{2^n} \vdash 2^{2^n}$
$\mathsf{transpose}_{2^n,1} := \mathsf{iden} : 2^{2^n} \vdash 2^{2^n}$

B.1.1.31 find-first-high

(CAUTION: Not defined when $n = 0$.)
$\ulcorner \mathsf{'find\text{-}first\text{-}high'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 31 \urcorner \cdot \ulcorner n \urcorner$

$\mathsf{find\text{-}first\text{-}high}_{2^n} : 2^{2^n} \vdash 2^{2^{\lceil \mathrm{lb}(n+1) \rceil}}$ for $1 \leq n$

B.1.1.32 find-last-high

(CAUTION: Not defined when $n = 0$.)
$\ulcorner \text{'}\mathsf{find\text{-}last\text{-}high}\text{'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 32 \urcorner \cdot \ulcorner n \urcorner$
$\mathsf{find\text{-}last\text{-}high}_{2^n} : 2^{2^n} \vdash 2^{2^{\lceil \mathrm{lb}(n+1) \rceil}}$ for $1 \leq n$

B.1.1.33 bit

(CAUTION: Not defined when $n \leq 1$.)
$\ulcorner \text{'}\mathsf{bit}\text{'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 33 \urcorner \cdot \ulcorner n - 1 \urcorner$
$\mathsf{bit}_{2^n} : 2^{2^{\lceil \mathrm{lb}(n) \rceil}} \times 2^{2^n} \vdash 2$ for $2 \leq n$

Aliases:
$\mathsf{bit}_1 := \mathsf{drop} : \mathbb{1} \times 2 \vdash 2$
$\mathsf{bit}_2 := \mathsf{ch}_1 : 2 \times 2^2 \vdash 2$

B.1.2 110100...: Jets for arithmetic

B.1.2.1 one

(CAUTION: Not defined when $n = 0$. See Aliases.)
$\ulcorner \text{'}\mathsf{one}\text{'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner n \urcorner$
$\mathsf{one}_{2^n} : \mathbb{1} \vdash 2^{2^n}$ for $1 \leq n$

Properties:
$\lceil \mathrm{bit}_{2^n,i}(\llbracket \mathsf{one}_{2^n} \rrbracket(x)) \rceil_1 = 0$ when $0 \leq i < 2^n - 1$
$\lceil \mathrm{bit}_{2^n,2^n-1}(\llbracket \mathsf{one}_{2^n} \rrbracket(x)) \rceil_1 = 1$
$\lceil \llbracket \mathsf{one}_{2^n} \rrbracket\langle\rangle \rceil_{2^n} = 1$

Aliases:
$\mathsf{one}_1 := \mathsf{high}_1 : \mathbb{1} \vdash 2$

B.1.2.2 full-add

(Note: $\text{'}\mathsf{full\text{-}add}\text{'}_1$ is composed from ‘maj‘ and ‘tri-xor‘.)
$\ulcorner \text{'}\mathsf{full\text{-}add}\text{'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 2 \urcorner \cdot \ulcorner n \urcorner$
$\mathsf{full\text{-}add}_{2^n} : 2 \times (2^{2^n} \times 2^{2^n}) \vdash 2 \times 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil \llbracket \mathsf{full\text{-}add}_{2^n} \rrbracket\langle c, \langle x, y \rangle\rangle \rceil_{1,2^n} = \lceil x \rceil_{2^n} + \lceil y \rceil_{2^n} + \lceil c_1 \rceil_1$

B.1.2.3 add

$\ulcorner \text{'}\mathsf{add}\text{'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 3 \urcorner \cdot \ulcorner n + 1 \urcorner$
$\mathsf{add}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2 \times 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil \llbracket \mathsf{add}_{2^n} \rrbracket\langle x, y \rangle \rceil_{1,2^n} = \lceil x \rceil_{2^n} + \lceil y \rceil_{2^n}$

Aliases:
$\mathsf{popcount}_2 := \mathsf{add}_1 : 2^2 \vdash 2^2$

B.1.2.4 full-increment

(CAUTION: Not defined when $n = 0$. See Aliases.)

$\ulcorner$'full-increment'$_{2^n}\urcorner := [110]_{\mathbb{2}}\cdot\ulcorner 2\urcorner\cdot\ulcorner 4\urcorner\cdot\ulcorner n\urcorner$
$\mathsf{full\text{-}increment}_{2^n} : \mathbb{2}\times\mathbb{2}^{2^n} \vdash \mathbb{2}\times\mathbb{2}^{2^n}$ for $1\leq n$

Properties:
$\lceil[\![\mathsf{full\text{-}increment}_{2^n}]\!]\langle c, x\rangle\rceil_{1,2^n} = \lceil x\rceil_{2^n} + \lceil c_1\rceil_1$

B.1.2.5 increment

$\ulcorner$'increment'$_{2^n}\urcorner := [110]_{\mathbb{2}}\cdot\ulcorner 2\urcorner\cdot\ulcorner 5\urcorner\cdot\ulcorner n\urcorner$
$\mathsf{increment}_{2^n} : \mathbb{2}^{2^n} \vdash \mathbb{2}\times\mathbb{2}^{2^n}$ for $0\leq n$

Properties:
$\lceil[\![\mathsf{increment}_{2^n}]\!](x)\rceil_{1,2^n} = \lceil x\rceil_{2^n} + 1$

B.1.2.6 popcount

(CAUTION: Not defined when $n\leq 1$.)
$\ulcorner$'popcount'$_{2^n}\urcorner := [110]_{\mathbb{2}}\cdot\ulcorner 2\urcorner\cdot\ulcorner 6\urcorner\cdot\ulcorner n-1\urcorner$
$\mathsf{popcount}_{2^n} : \mathbb{2}^{2^n} \vdash 2^{2^{\lceil \mathrm{lb}(n+1)\rceil}}$ for $1\leq n$

Properties:
$\lceil[\![\mathsf{popcount}_{2^n}]\!](x)\rceil_{2^{\lceil \mathrm{lb}(n+1)\rceil}} = \sum_{i=0}^{2^n-1} \lceil \mathrm{bit}_{2^n,i}(x)\rceil_1$

Aliases:
$\mathsf{popcount}_1 := \mathsf{iden} : \mathbb{2} \vdash \mathbb{2}$
$\mathsf{popcount}_2 := \mathsf{add}_1 : \mathbb{2}^2 \vdash 2^2$

B.1.2.7 full-subtract

(Note: composition of full-add with complement applied to 1st and 3rd arguments).
$\ulcorner$'full-subtract'$_{2^n}\urcorner := [110]_{\mathbb{2}}\cdot\ulcorner 2\urcorner\cdot\ulcorner 7\urcorner\cdot\ulcorner n\urcorner$
$\mathsf{full\text{-}subtract}_{2^n} : \mathbb{2}\times(\mathbb{2}^{2^n}\times\mathbb{2}^{2^n}) \vdash \mathbb{2}\times\mathbb{2}^{2^n}$ for $0\leq n$

Properties:
$\lceil[\![\mathsf{full\text{-}subtract}_{2^n}]\!]\langle c, \langle x, y\rangle\rangle\rceil_{1,2^n} \equiv \lceil x\rceil_{2^n} - \lceil y\rceil_{2^n} - \lceil c_1\rceil_1 (\mathrm{mod}\, 2^{2^n+1})$

B.1.2.8 subtract

$\ulcorner$'subtract'$_{2^n}\urcorner := [110]_{\mathbb{2}}\cdot\ulcorner 2\urcorner\cdot\ulcorner 8\urcorner\cdot\ulcorner n\urcorner$
$\mathsf{subtract}_{2^n} : \mathbb{2}^{2^n}\times\mathbb{2}^{2^n} \vdash \mathbb{2}\times\mathbb{2}^{2^n}$ for $0\leq n$

Properties:
$\lceil[\![\mathsf{full\text{-}subtract}_{2^n}]\!]\langle x, y\rangle\rceil_{1,2^n} \equiv \lceil x\rceil_{2^n} - \lceil y\rceil_{2^n} (\mathrm{mod}\, 2^{2^n+1})$

B.1.2.9 negate

(CAUTION: Not defined when $n=0$.)
$\ulcorner$'negate'$_{2^n}\urcorner := [110]_{\mathbb{2}}\cdot\ulcorner 2\urcorner\cdot\ulcorner 9\urcorner\cdot\ulcorner n\urcorner$
$\mathsf{negate}_{2^n} : \mathbb{2}^{2^n} \vdash \mathbb{2}\times\mathbb{2}^{2^n}$ for $1\leq n$

Properties:
$\lceil[\![\mathsf{negate}_{2^n}]\!](x)\rceil_{1,2^n} \equiv -\lceil x\rceil_{2^n} (\mathrm{mod}\, 2^{2^n+1})$

B.1.2.10 full-decrement

$\ulcorner$'full-decrement'$_{2^n}\urcorner := [110]_2 . \ulcorner 2\urcorner . \ulcorner 10\urcorner . \ulcorner n\urcorner$
full-decrement$_{2^n} : 2 \times 2^{2^n} \vdash 2 \times 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil [\![\text{full-decrement}_{2^n}]\!] \langle c, x \rangle \rceil_{1,2^n} \equiv \lceil x \rceil_{2^n} - \lceil c_1 \rceil_1 (\mathrm{mod}\, 2^{2^n+1})$

B.1.2.11 decrement

$\ulcorner$'decrement'$_{2^n}\urcorner := [110]_2 . \ulcorner 2\urcorner . \ulcorner 11\urcorner . \ulcorner n\urcorner$
decrement$_{2^n} : 2^{2^n} \vdash 2 \times 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil [\![\text{decrement}_{2^n}]\!] (x) \rceil_{1,2^n} \equiv \lceil x \rceil_{2^n} - 1 (\mathrm{mod}\, 2^{2^n+1})$

B.1.2.12 full-multiply

(Note: 'full-multiply'$_1$ is composed from 'full-add'$_1$ and 'and'.)
$\ulcorner$'full-multiply'$_{2^n}\urcorner := [110]_2 . \ulcorner 2\urcorner . \ulcorner 12\urcorner . \ulcorner n\urcorner$
full-multiply$_{2^n} : (2^{2^n} \times 2^{2^n}) \times (2^{2^n} \times 2^{2^n}) \vdash 2^{2^{n+1}}$ for $0 \leq n$

Properties:
$\lceil [\![\text{full-multiplier}_{2^n}]\!] \langle \langle c_1, c_2 \rangle, \langle x, y \rangle \rangle \rceil_{2^{n+1}} = \lceil x \rceil_{2^n} \cdot \lceil y \rceil_{2^n} + \lceil c_1 \rceil_{2^n} + \lceil c_2 \rceil_{2^n}$

B.1.2.13 multiply

$\ulcorner$'multiply'$_{2^n}\urcorner := [110]_2 . \ulcorner 2\urcorner . \ulcorner 13\urcorner . \ulcorner n\urcorner$
multiply$_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^{n+1}}$ for $0 \leq n$

Properties:
$\lceil [\![\text{multiply}_{2^n}]\!] \langle x, y \rangle \rceil_{2^{n+1}} = \lceil x \rceil_{2^n} \cdot \lceil y \rceil_{2^n}$

B.1.2.14 is-zero

(CAUTION: Not defined when $n = 0$. See Aliases.)
(Note: complement of some).
$\ulcorner$'is-zero'$_{2^n}\urcorner := [110]_2 . \ulcorner 2\urcorner . \ulcorner 14\urcorner . \ulcorner n\urcorner$

Properties:
$\lceil [\![\text{is-zero}_{2^n}]\!] (x) \rceil_1 = \chi(\lceil x \rceil_{2^n} = 0)$

B.1.2.15 is-one

(CAUTION: Not defined when $n = 0$. See Aliases.)
$\ulcorner$'is-one'$_{2^n}\urcorner := [110]_2 . \ulcorner 2\urcorner . \ulcorner 15\urcorner . \ulcorner n\urcorner$

Properties:
$\lceil [\![\text{is-one}_{2^n}]\!] (x) \rceil_1 = \chi(\lceil x \rceil_{2^n} = 1)$

B.1.2.16 le (unsigned)

(Note: borrow bit from subtractor)
$\ulcorner$'le'$_{2^n}\urcorner := [110]_2 . \ulcorner 2\urcorner . \ulcorner 16\urcorner . \ulcorner n + 1\urcorner$
le$_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2$ for $0 \leq n$

Properties:
$\lceil [\![\text{le}_{2^n}]\!] \langle x, y \rangle \rceil_1 = \chi(\lceil x \rceil_{2^n} \leq \lceil y \rceil_{2^n})$

B.1.2.17 lt (unsigned)

$\ulcorner \mathsf{'lt'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 17 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{lt}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2$ for $0 \leq n$

Properties:
$\lceil [\![\mathsf{lt}_{2^n}]\!] \langle x, y \rangle \rceil_1 = \chi(\lceil x \rceil_{2^n} < \lceil y \rceil_{2^n})$

B.1.2.18 min (unsigned)

$\ulcorner \mathsf{'min'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 18 \urcorner \cdot \ulcorner n \urcorner$
$\mathsf{min}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n}$ for $1 \leq n$

Properties:
$z \leq \lceil [\![\mathsf{'min'}_{2^n}]\!] \langle x, y \rangle \rceil_{2^n}$ if and only if $z \leq \lceil x \rceil_{2^n}$ and $z \leq \lceil y \rceil_{2^n}$

Aliases:
$\mathsf{min}_1 := \mathsf{and}_1 : 2 \times 2 \vdash 2$

B.1.2.19 max (unsigned)

$\ulcorner \mathsf{'max'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 19 \urcorner \cdot \ulcorner n \urcorner$
$\mathsf{max}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n}$ for $1 \leq n$

Properties:
$\lceil [\![\mathsf{max}_{2^n}]\!] \langle x, y \rangle \rceil_{2^n} \leq z$ if and only if $\lceil x \rceil_{2^n} \leq z$ and $\lceil y \rceil_{2^n} \leq z$

Aliases:
$\mathsf{max}_1 := \mathsf{or}_1 : 2 \times 2 \vdash 2$

B.1.2.20 median (unsigned)

$\ulcorner \mathsf{'median'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 20 \urcorner \cdot \ulcorner n \urcorner$
$\mathsf{median}_{2^n} : 2^{2^n} \times (2^{2^n} \times 2^{2^n}) \vdash 2^{2^n}$ for $1 \leq n$

Properties:
$\lceil [\![\mathsf{min}_{2^n}]\!] \langle x_i, x_j \rangle \rceil_{2^n} \leq \lceil [\![\mathsf{median}_{2^n}]\!] \langle x_1, \langle x_2, x_3 \rangle \rangle \rceil_{2^n} \leq \lceil [\![\mathsf{max}_{2^n}]\!] \langle x_i, x_j \rangle \rceil_{2^n}$ for all $1 \leq i < j \leq 3$.

Aliases:
$\mathsf{median}_1 := \mathsf{maj}_1 : 2 \times (2 \times 2) \vdash 2$

B.1.2.21 div2n1n

(Note: helper function for divmod. See Fast recursive division by Bunikel and Ziegler (1998).)
$\ulcorner \mathsf{'div2n1n'}_{2^n} \urcorner := [110]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 21 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{div2n1n}_{2^n} : 2^{2^{n+1}} \times 2^{2^n} \vdash 2^{2^n} \times 2^{2^n}$ for $0 \leq n$

Properties:
When $2^{n-1} \leq \lceil y \rceil_{2^n}$ and $\lceil x \rceil_{2^{n+1}} < 2^n \cdot \lceil y \rceil_{2^n}$,

a) $\lceil y \rceil_{2^n} \cdot \lceil \pi_1([\![\mathsf{div2n1n}_{2^n}]\!] \langle x, y \rangle) \rceil_{2^n} = \lceil x \rceil_{2^{n+1}} - \lceil \pi_2([\![\mathsf{div2n1n}_{2^n}]\!] \langle x, y \rangle) \rceil_{2^n}$, and

b) $\lceil \pi_2([\![\mathsf{div2n1n}_{2^n}]\!] \langle x, y \rangle) \rceil_{2^n} < \lceil y \rceil_{2^n}$.

When $2^{n-1} > \lceil y \rceil_{2^n}$ or $\lceil x \rceil_{2^{n+1}} \geqslant 2^n \cdot \lceil y \rceil_{2^n}$, then $\mathrm{bit}_{2^{2n},i}(\llbracket \mathsf{div2n1n}_{2^n} \rrbracket \langle x, y \rangle) = 1$.

B.1.2.22 div-mod (unsigned)

$\ulcorner \text{'div-mod'}_{2^n} \urcorner := \mathtt{[110]}_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 22 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{div\text{-}mod}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n} \times 2^{2^n}$ for $0 \leq n$

Properties:
$\lceil y \rceil_{2^n} \cdot \lceil \pi_1(\llbracket \mathsf{div\text{-}mod}_{2^n} \rrbracket \langle x, y \rangle) \rceil_{2^n} = \lceil x \rceil_{2^n} - \lceil \pi_2(\llbracket \mathsf{div\text{-}mod}_{2^n} \rrbracket \langle x, y \rangle) \rceil_{2^n}$

For all $r \in \mathbb{N}$ such that $\lceil y \rceil_{2^n} | \lceil x \rceil_{2^n} - r$ we have $\lceil \pi_2(\llbracket \mathsf{div\text{-}mod}_{2^n} \rrbracket \langle x, y \rangle) \rceil_{2^n} \leq r$.

B.1.2.23 divide (unsigned)

$\ulcorner \text{'divide'}_{2^n} \urcorner := \mathtt{[110]}_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 23 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{divide}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n}$ for $0 \leq n$

B.1.2.24 modulo (unsigned)

$\ulcorner \text{'modulo'}_{2^n} \urcorner := \mathtt{[110]}_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 24 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{modulo}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n}$ for $0 \leq n$

B.1.2.25 divides (unsigned)

$\ulcorner \text{'divides'}_{2^n} \urcorner := \mathtt{[110]}_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 25 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{divides}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2$ for $0 \leq n$

Properties:
$\lceil \llbracket \mathsf{divides}_{2^n} \rrbracket \langle x, y \rangle \rceil_1 = \chi(\lceil x \rceil_{2^n} | \lceil y \rceil_{2^n})$

B.1.2.26 eea (unsigned)

$\ulcorner \text{'eea'}_{2^n} \urcorner := \mathtt{[110]}_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 26 \urcorner \cdot \ulcorner n+1 \urcorner$
$\mathsf{eea}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash (((2^{2^n} \times 2^{2^n}) + (2^{2^n} \times 2^{2^n})) \times (2^{2^n} \times 2^{2^n})) \times 2^{2^n}$ for $0 \leq n$

Properties:
Let $\langle \langle b, \langle c_y, c_x \rangle \rangle, d \rangle := \llbracket \mathsf{eea}_{2^n} \rrbracket \langle x, y \rangle$

$\lceil c_x \rceil_{2^n} \lceil d \rceil_{2^n} = \lceil x \rceil_{2^n}$
$\lceil c_y \rceil_{2^n} \lceil d \rceil_{2^n} = \lceil y \rceil_{2^n}$

If $\sigma^{\mathbf{L}} \langle s, t \rangle = b$ then $\lceil s \rceil_{2^n} \lceil x \rceil_{2^n} - \lceil t \rceil_{2^n} \lceil y \rceil_{2^n} = \lceil d \rceil_{2^n}$ and either $\lceil s \rceil_{2^n} < \frac{\lceil c_y \rceil_{2^n}}{2}$ or $\lceil t \rceil_{2^n} < \frac{\lceil c_x \rceil_{2^n}}{2}$.
If $\sigma^{\mathbf{R}} \langle s, t \rangle = b$ then $\lceil s \rceil_{2^n} \lceil x \rceil_{2^n} - \lceil t \rceil_{2^n} \lceil y \rceil_{2^n} = -\lceil d \rceil_{2^n}$ and either $\lceil s \rceil_{2^n} < \frac{\lceil c_y \rceil_{2^n}}{2}$ or $\lceil t \rceil_{2^n} < \frac{\lceil c_x \rceil_{2^n}}{2}$.

$\llbracket \mathsf{eea}_{2^n} \rrbracket \langle \lfloor x \rfloor, \lfloor x \rfloor \rangle = \langle \langle \sigma^{\mathbf{L}} \langle \lfloor 1 \rfloor, \lfloor 0 \rfloor \rangle, \langle \lfloor 1 \rfloor, \lfloor 1 \rfloor \rangle \rangle, \lfloor x \rfloor \rangle$.

B.1.2.27 bezout (unsigned)

(CAUTION: Not defined when $n \leq 1$.)
$\ulcorner \text{'bezout'}_{2^n} \urcorner := \mathtt{[110]}_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 27 \urcorner \cdot \ulcorner n \urcorner$
$\mathsf{bezout}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash (2^{2^n} \times 2^{2^n}) + (2^{2^n} \times 2^{2^n})$ for $1 \leq n$

B.1.2.28 gcd (unsigned)

(CAUTION: Not defined when $n \le 1$.)
$\ulcorner$'gcd'$_{2^n}\urcorner := [110]_2 \cdot \ulcorner 2\urcorner \cdot \ulcorner 28\urcorner \cdot \ulcorner n\urcorner$
$\mathsf{gcd}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n}$ for $1 \le n$

B.1.2.29 cofactors (unsigned)

(CAUTION: Not defined when $n \le 1$.)
$\ulcorner$'cofactors'$_{2^n}\urcorner := [110]_2 \cdot \ulcorner 2\urcorner \cdot \ulcorner 29\urcorner \cdot \ulcorner n\urcorner$
$\mathsf{cofactors}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^n} \times 2^{2^n}$ for $1 \le n$

B.1.2.30 lcm (unsigned)

$\ulcorner$'lcm'$_{2^n}\urcorner := [110]_2 \cdot \ulcorner 2\urcorner \cdot \ulcorner 30\urcorner \cdot \ulcorner n+1\urcorner$
$\mathsf{lcm}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^{2^{n+1}}$ for $0 \le n$

B.1.2.31 jacobi (unsigned)

$\ulcorner$'jacobi'$_{2^n}\urcorner := [110]_2 \cdot \ulcorner 2\urcorner \cdot \ulcorner 31\urcorner \cdot \ulcorner n+1\urcorner$
$\mathsf{jacobi}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2^2$ for $0 \le n$

B.1.2.32 absolute-value (signed input/unsigned output)

$\ulcorner$'absolute-value'$_{2^n}\urcorner := [110]_2 \cdot \ulcorner 2\urcorner \cdot \ulcorner 32\urcorner \cdot \ulcorner n\urcorner$
$\mathsf{absolute\text{-}value}_{2^n} : 2^{2^n} \vdash 2^{2^n}$ for $0 \le n$

B.1.2.33 sign

(CAUTION: Not defined when $n = 0$. See Aliases.)
$\ulcorner$'sign'$_{2^n}\urcorner := [110]_2 \cdot \ulcorner 2\urcorner \cdot \ulcorner 33\urcorner \cdot \ulcorner n\urcorner$
$\mathsf{sign}_{2^n} : 2^{2^n} \vdash 2^2$ for $1 \le n$

Properties:
$[\![\mathsf{sign}_{2^n}]\!](x) = \langle [\![\mathsf{leftmost}_{2^n}]\!](x), [\![\mathsf{some}_{2^n}]\!](x) \rangle$

B.1.2.34 signed-le

$\ulcorner$'signed-le'$_{2^n}\urcorner := [110]_2 \cdot \ulcorner 2\urcorner \cdot \ulcorner 34\urcorner \cdot \ulcorner n+1\urcorner$
$\mathsf{signed\text{-}le}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2$ for $0 \le n$

B.1.2.35 signed-lt

$\ulcorner$'signed-lt'$_{2^n}\urcorner := [110]_2 \cdot \ulcorner 2\urcorner \cdot \ulcorner 35\urcorner \cdot \ulcorner n+1\urcorner$
$\mathsf{signed\text{-}lt}_{2^n} : 2^{2^n} \times 2^{2^n} \vdash 2$ for $0 \le n$

B.1.2.36 signed-min

B.1.2.37 signed-max

B.1.2.38 signed-median

B.1.2.39 signed-right-shift

Right shift by a signed amount with sign extension. Negative values are a left shift.

B.1.2.40 signed-divmod (unsigned denominator)

B.1.2.41 signed-div (unsigned denominator)

B.1.2.42 signed-signed-divmod (signed denominator)

B.1.2.43 signed-signed-div (signed denominator)

B.1.3 110101...: Jets for hash functions

B.1.3.1 1101010...: Jets for SHA-2

In this section we define $\mathrm{Ctx8} := (2^8)^{<64} \times (2^{64} \times 2^{256})$. This represent a type of SHA-256 contexts for hashing variable length byte strings. The first component is a buffer of bytes representing a partial block. The second component is the number of times the compression function has been invoked. The third component is the SHA-256 midstate of compressed data so for (excluding the data in the partial block).

Note that SHA-256 only allows the compression counter to be less than 2^{55}. The jets below will fail if the compression counter is 2^{55} or larger, or if executing the function would cause the counter to become 2^{55} or larger.

B.1.3.1.1 sha-256-block

$\ulcorner$'sha-256-block'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 1 \urcorner$
sha-256-block : $2^{256} \times 2^{512} \vdash 2^{256}$

B.1.3.1.2 sha-256-iv

$\ulcorner$'sha-256-iv'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 2 \urcorner$
sha-256-iv : $\mathbb{1} \vdash 2^{256}$

B.1.3.1.3 sha-256-ctx-8-add

$\ulcorner$'sha-256-ctx-8-add'$_{2^n}\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 3 \urcorner \cdot \ulcorner n+1 \urcorner$
sha-256-ctx-8-add$_{2^n}$: $\mathrm{Ctx8} \times (2^8)^{2^n} \vdash \mathrm{Ctx8}$

B.1.3.1.4 sha-256-ctx-8-add-buffer-511

$\ulcorner$'sha-256-ctx-8-add-buffer'$_{511}\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 4 \urcorner$
sha-256-ctx-8-add-buffer$_{511}$: $\mathrm{Ctx8} \times (2^8)^{<512} \vdash \mathrm{Ctx8}$

B.1.3.1.5 sha-256-ctx-8-finalize

$\ulcorner$'sha-256-ctx-8-finalize'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 5 \urcorner$
sha-256-ctx-8-finalize : $\mathrm{Ctx8} \vdash 2^{256}$

B.1.3.1.6 sha-256-ctx-8-init

$\ulcorner$'sha-256-ctx-8-init'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 6 \urcorner$
sha-256-ctx-8-init : $\mathbb{1} \vdash \mathrm{Ctx8}$

B.1.3.2 110101100...: Jets for SHA-3

In this section we define $X^5 := X \times X^4$ and $X^{1600} := ((X^{64})^5)^5$.

B.1.3.2.1 sha3-zero

⌜'sha3-zero'⌝ := $[110]_2$·⌜3⌝·⌜2⌝·⌜1⌝
sha3-zero : $\mathbb{1} \vdash 2^{1600}$

B.1.3.2.2 sha3-absorb

(Note: we should proably byte-stwap the input before xoring it into place
⌜'sha3-absorb'$_{n,m}$⌝ := $[110]_2$·⌜3⌝·⌜2⌝·⌜2⌝·⌜n⌝·⌜m⌝
sha3-absorb$_{n,m}$: $2^{64} \times 2^{1600} \vdash 2^{1600}$ for $1 \leq n, m \leq 5$

B.1.3.2.3 sha3-xor

⌜'sha3-xor'⌝ := $[110]_2$·⌜3⌝·⌜2⌝·⌜3⌝
sha3-xor : $2^{1600} \times 2^{1600} \vdash 2^{1600}$

B.1.3.2.4 sha3-permute

⌜'sha3-permute'⌝ := $[110]_2$·⌜3⌝·⌜2⌝·⌜4⌝
sha3-permute: $2^{1600} \vdash 2^{1600}$

B.1.3.2.5 sha3-squeeze-256

⌜'sha3-squeeze-256'⌝ := $[110]_2$·⌜3⌝·⌜2⌝·⌜5⌝
sha3-squeeze-256: $2^{1600} \vdash 2^{256}$

B.1.3.2.6 sha3-squeeze-512

⌜'sha3-squeeze-512'⌝ := $[110]_2$·⌜3⌝·⌜2⌝·⌜6⌝
sha3-squeeze-512: $2^{1600} \vdash 2^{512}$

B.1.3.3 110101101...: Jets for RIPEMD

B.1.3.4 110101110000...: Jets for SHA-1 (RESERVED)

B.1.4 110110000...: Jets for elliptic curve functions

B.1.4.1 1101100000...: Jets for secp256k1

In this section we define Scalar := 2^{256}, $2^{129} := 2 \times 2^{128}$, FE := 2^{256}, GE := FE × FE, and GEJ := GE × FE, Point := $2 \times$ FE.

(Note: To convert GE to GEJ pair with one).

TODO: Verify that all *equivalent* FE and Scalar inputs yield *equal* outputs.

B.1.4.1.1 secp256k1-point-verify

(Note: Support only recommened up to $2^n \leq$ TODO 8?.)
⌜'secp256k1-point-verify'$_{2^n}$⌝ := $[110]_2$·⌜4⌝·⌜1⌝·⌜1⌝·⌜$n+1$⌝
secp256k1-point-verify$_{2^n}$: $((\text{Scalar} \times \text{Point})^{2^n} \times \text{Scalar}) \times \text{Point} \vdash \mathbb{1}$ for $0 \leq n$

B.1.4.1.2 secp256k1-decompress

⌜'secp256k1-decompress'⌝ := $[110]_2$·⌜4⌝·⌜1⌝·⌜2⌝·⌜$n+1$⌝
secp256k1-decompress: Point $\vdash$ S(GE) for $0 \leq n$

B.1.4.1.3 secp256k1-linear-verify

(Note: Support only recommened up to $2^n \leq$ TODO 8?.)
⌜'secp256k1-linear-verify'$_{2^n}$⌝ := $[110]_2$·⌜4⌝·⌜1⌝·⌜3⌝·⌜$n+1$⌝

secp256k1-linear-verify$_{2^n}$: $((\mathrm{Scalar} \times \mathrm{GE})^{2^n} \times \mathrm{Scalar}) \times \mathrm{GE} \vdash \mathbb{1}$ for $0 \leq n$

B.1.4.1.4 secp256k1-linear-combination

(Note: Support only recommened up to $2^n \leq$ TODO 8?.)

$\ulcorner$'secp256k1-linear-combination'$_{2^n}\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 4 \urcorner \cdot \ulcorner n+1 \urcorner$

secp256k1-linear-combination$_{2^n}$: $(\mathrm{Scalar} \times \mathrm{GEJ})^{2^n} \times \mathrm{Scalar} \vdash \mathrm{GEJ}$ for $0 \leq n$

B.1.4.1.5 secp256k1-scale

$\ulcorner$'secp256k1-scale'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 5 \urcorner$

secp256k1-scale: $\mathrm{Scalar} \times \mathrm{GEJ} \vdash \mathrm{GEJ}$

B.1.4.1.6 secp256k1-generate

$\ulcorner$'secp256k1-generate'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 6 \urcorner$

secp256k1-generate: $\mathrm{Scalar} \vdash \mathrm{GEJ}$

B.1.4.1.7 secp256k1-gej-infinity

$\ulcorner$'secp256k1-gej-infinity'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 7 \urcorner$

secp256k1-gej-infinity: $\mathbb{1} \vdash \mathrm{GEJ}$

B.1.4.1.8 secp256k1-gej-normalize

$\ulcorner$'secp256k1-gej-normalize'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 8 \urcorner$

secp256k1-gej-normalize: $\mathrm{GEJ} \vdash \mathrm{S}(\mathrm{GE})$

B.1.4.1.9 secp256k1-gej-negate

$\ulcorner$'secp256k1-gej-negate'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 9 \urcorner$

secp256k1-gej-negate: $\mathrm{GEJ} \vdash \mathrm{GEJ}$

B.1.4.1.10 secp256k1-ge-negate

$\ulcorner$'secp256k1-ge-negate'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 10 \urcorner$

secp256k1-ge-negate: $\mathrm{GE} \vdash \mathrm{GE}$

B.1.4.1.11 secp256k1-gej-double

$\ulcorner$'secp256k1-gej-double'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 11 \urcorner$

secp256k1-gej-double: $\mathrm{GEJ} \vdash \mathrm{GEJ}$

B.1.4.1.12 secp256k1-gej-add

$\ulcorner$'secp256k1-gej-add'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 12 \urcorner$

secp256k1-gej-add: $\mathrm{GEJ} \times \mathrm{GEJ} \vdash \mathrm{GEJ}$

B.1.4.1.13 secp256k1-gej-ge-add-ex

$\ulcorner$'secp256k1-gej-ge-add-ex'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 13 \urcorner$

secp256k1-gej-ge-add-ex: $\mathrm{GEJ} \times \mathrm{GE} \vdash \mathrm{FE} \times \mathrm{GEJ}$

B.1.4.1.14 secp256k1-gej-ge-add

⌜'secp256k1-gej-ge-add'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜14⌝
secp256k1-gej-ge-add: GEJ × GE ⊢ GEJ

B.1.4.1.15 secp256k1-gej-rescale

⌜'secp256k1-gej-rescale'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜15⌝
secp256k1-gej-rescale: GEJ × FE ⊢ GEJ

B.1.4.1.16 secp256k1-gej-is-infinity

⌜'secp256k1-gej-is-infinity'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜16⌝
secp256k1-gej-is-infinity: GEJ ⊢ 2

B.1.4.1.17 secp256k1-gej-equiv Does not exist in libsecp256k1

Warning: this can be implemented by comparing coordinates (by cross multiplication) or by adding points and testing for infinity. However these two implementations yeilds different results for off curve points.
⌜'secp256k1-gej-equiv'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜17⌝
secp256k1-gej-equiv: GEJ × GEJ ⊢ 2

B.1.4.1.18 secp256k1-gej-ge-equiv Does not exist in libsecp256k1

Warning: this can be implemented by comparing coordinates (by cross multiplication) or by adding points and testing for infinity. However these two implementations yeilds different results for off curve points.

⌜'secp256k1-gej-ge-equiv'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜18⌝
secp256k1-gej-ge-equiv: GEJ × GE ⊢ 2

B.1.4.1.19 secp256k1-gej-x-equiv

⌜'secp256k1-gej-x-equiv'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜19⌝
secp256k1-gej-x-equiv: FE × GEJ ⊢ 2

B.1.4.1.20 secp256k1-gej-y-is-odd

⌜'secp256k1-gej-y-is-odd'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜20⌝
secp256k1-gej-y-is-odd: GEJ ⊢ 2

B.1.4.1.21 secp256k1-gej-is-on-curve

⌜'secp256k1-gej-is-on-curve'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜21⌝
secp256k1-gej-is-on-curve: GEJ ⊢ 2

B.1.4.1.22 secp256k1-ge-is-on-curve

⌜'secp256k1-ge-is-on-curve'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜22⌝
secp256k1-ge-is-on-curve: GEJ ⊢ 2

B.1.4.1.23 secp256k1-scalar-normalize

⌜'secp256k1-scalar-normalize'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜23⌝
secp256k1-scalar-normalize: Scalar ⊢ Scalar

B.1.4.1.24 secp256k1-scalar-negate

⌜'secp256k1-scalar-negate'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜24⌝

'secp256k1-scalar-negate': $\mathrm{Scalar} \vdash \mathrm{Scalar}$

B.1.4.1.25 secp256k1-scalar-add

$\ulcorner$'secp256k1-scalar-add'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 25 \urcorner$
secp256k1-scalar-add: $\mathrm{Scalar} \times \mathrm{Scalar} \vdash \mathrm{Scalar}$

B.1.4.1.26 secp256k1-scalar-square

$\ulcorner$'secp256k1-scalar-square'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 26 \urcorner$
secp256k1-scalar-square: $\mathrm{Scalar} \vdash \mathrm{Scalar}$

B.1.4.1.27 secp256k1-scalar-multiply

$\ulcorner$'secp256k1-scalar-multiply'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 27 \urcorner$
secp256k1-scalar-multiply: $\mathrm{Scalar} \times \mathrm{Scalar} \vdash \mathrm{Scalar}$

B.1.4.1.28 secp256k1-scalar-multiply-lambda

$\ulcorner$'secp256k1-scalar-multiply-lambda'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 28 \urcorner$
secp256k1-scalar-multiply-lambda: $\mathrm{Scalar} \vdash \mathrm{Scalar}$

B.1.4.1.29 secp256k1-scalar-invert

$\ulcorner$'secp256k1-scalar-invert'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 29 \urcorner$
secp256k1-scalar-invert: $\mathrm{Scalar} \vdash \mathrm{Scalar}$

B.1.4.1.30 secp256k1-scalar-is-zero

$\ulcorner$'secp256k1-scalar-is-zero'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 30 \urcorner$
secp256k1-scalar-is-zero: $\mathrm{Scalar} \vdash 2$

B.1.4.1.31 secp256k1-ge-scale-lambda

$\ulcorner$'secp256k1-ge-scale-lambda'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 31 \urcorner$
secp256k1-ge-scale-lambda: $\mathrm{GE} \vdash \mathrm{GE}$

B.1.4.1.32 secp256k1-gej-scale-lambda Consider removing

$\ulcorner$'secp256k1-gej-scale-lambda'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 32 \urcorner$
secp256k1-gej-scale-lambda: $\mathrm{GEJ} \vdash \mathrm{GEJ}$

B.1.4.1.33 secp256k1-scalar-split-lambda

$\ulcorner$'secp256k1-scalar-split-lambda'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 33 \urcorner$
secp256k1-scalar-split-lambda: $\mathrm{Scalar} \vdash 2^{129} \times 2^{129}$

TODO Properties:
$\lambda \cdot \lceil \pi_1([\![\text{'secp256k1-scalar-split'}]\!](x)) \rceil_{2^{128}} + \lceil \pi_2([\![\text{'secp256k1-scalar-split'}]\!](x)) \rceil_{2^{128}} \equiv_{\mathrm{Scalar}} \lceil x \rceil_{2^{256}}$

B.1.4.1.34 secp256k1-short-scalar

$\ulcorner$'secp256k1-short-scalar'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 34 \urcorner$
secp256k1-short-scalar: $2^{129} \vdash \mathrm{Scalar}$

B.1.4.1.35 secp256k1-fe-normalize

⌜'secp256k1-fe-normalize'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜35⌝
secp256k1-fe-normalize: $\mathrm{FE} \vdash \mathrm{FE}$

B.1.4.1.36 secp256k1-fe-negate

⌜'secp256k1-fe-negate'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜36⌝
secp256k1-fe-negate: $\mathrm{FE} \vdash \mathrm{FE}$

B.1.4.1.37 secp256k1-fe-add

⌜'secp256k1-fe-negate'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜37⌝
secp256k1-fe-negate: $\mathrm{FE} \times \mathrm{FE} \vdash \mathrm{FE}$

B.1.4.1.38 secp256k1-fe-square

⌜'secp256k1-fe-square'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜38⌝
secp256k1-fe-square: $\mathrm{FE} \vdash \mathrm{FE}$

B.1.4.1.39 secp256k1-fe-multiply

⌜'secp256k1-fe-multiply'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜39⌝
secp256k1-fe-multiply: $\mathrm{FE} \times \mathrm{FE} \vdash \mathrm{FE}$

B.1.4.1.40 secp256k1-fe-multiply-beta

⌜'secp256k1-fe-multiply-beta'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜40⌝
secp256k1-fe-multiply-beta: $\mathrm{FE} \vdash \mathrm{FE}$

B.1.4.1.41 secp256k1-fe-invert

⌜'secp256k1-fe-invert'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜41⌝
gisecp256k1-fe-invert: $\mathrm{FE} \vdash \mathrm{FE}$

B.1.4.1.42 secp256k1-fe-square-root

⌜'secp256k1-fe-square-root'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜42⌝
secp256k1-fe-square-root: $\mathrm{FE} \vdash \mathrm{S}(\mathrm{FE})$

B.1.4.1.43 secp256k1-fe-is-zero

⌜'secp256k1-fe-is-zero'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜43⌝
secp256k1-fe-is-zero: $\mathrm{FE} \vdash 2$

B.1.4.1.44 secp256k1-fe-is-odd

⌜'secp256k1-fe-is-odd'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜44⌝
secp256k1-fe-is-odd: $\mathrm{FE} \vdash 2$

B.1.4.1.45 secp256k1-fe-is-quad

⌜'secp256k1-fe-is-quad'⌝ := [110]$_2$·⌜4⌝·⌜1⌝·⌜45⌝

secp256k1-fe-is-quad: $\mathrm{FE} \vdash 2$

B.1.4.1.46 secp256k1-hash-to-curve

$\ulcorner$'secp256k1-hash-to-curve'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 46 \urcorner$
secp256k1-hash-to-curve: $2^{256} \vdash \mathrm{GE}$

B.1.4.1.47 secp256k1-swu

$\ulcorner$'secp256k1-swu'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 47 \urcorner$
secp256k1-swu: $\mathrm{FE} \vdash \mathrm{GE}$

B.1.5 110110001...: Jets for digital signatures

B.1.5.1 1101100010...: Jets for secp256k1 based digital signatures

In this section we define $\mathrm{Pubkey} := 2^{256}$, $\mathrm{Signature} := 2^{512}$.

B.1.5.1.1 check-sig-verify

(Note: this jet can fail.)
$\ulcorner$'check-sig-verify'$\urcorner := [110]_2 \cdot \ulcorner 5 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 1 \urcorner$
check-sig-verify: $(\mathrm{Pubkey} \times 2^{512}) \times \mathrm{Signature} \vdash \mathbb{1}$

B.1.5.1.2 bip-0340-verify

(Note: this jet can fail.)
$\ulcorner$'bip-0340-verify'$\urcorner := [110]_2 \cdot \ulcorner 5 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 2 \urcorner$
bip-0340-verify: $(\mathrm{Pubkey} \times 2^{256}) \times \mathrm{Signature} \vdash \mathbb{1}$

B.1.5.1.3 bip-0340-challenge-iv

$\ulcorner$'bip-0340-challenge-iv'$\urcorner := [110]_2 \cdot \ulcorner 5 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 3 \urcorner$
bip-0340-challenge-iv: $\mathbb{1} \vdash 2^{256}$

B.1.5.1.4 bip-0340-challenge-midstate

$\ulcorner$'bip-0340-challenge-midstate'$\urcorner := [110]_2 \cdot \ulcorner 5 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 4 \urcorner$
bip-0340-challenge-midstate: $\mathrm{FE} \times \mathrm{Pubkey} \vdash 2^{256}$

B.1.5.1.5 secp256k1-signature-unpack

$\ulcorner$'secp256k1-signtaure-unpack'$\urcorner := [110]_2 \cdot \ulcorner 5 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 5 \urcorner$
secp256k1-signature-unpack: $\mathrm{Signature} \vdash \mathrm{S}(\mathrm{FE} \times \mathrm{Scalar})$

B.1.5.1.6 secp256k1-pubkey-unpack

$\ulcorner$'secp256k1-pubkey-unpack'$\urcorner := [110]_2 \cdot \ulcorner 5 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 6 \urcorner$
secp256k1-pubkey-unpack: $\mathrm{Pubkey} \vdash \mathrm{S}(\mathrm{GE})$

B.1.5.1.7 secp256k1-pubkey-unpack-neg

$\ulcorner$'secp256k1-pubkey-unpack-neg'$\urcorner := [110]_2 \cdot \ulcorner 5 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 7 \urcorner$
secp256k1-pubkey-unpack-neg: $\mathrm{Pubkey} \vdash \mathrm{S}(\mathrm{GE})$

B.1.5.1.8 secp256k1-ecdsa

B.1.6 110110010...: Jets for Simplicity

Tagged hash IVs for basic simplicity combinators, signature hashes, etc. The CMR of various expressions for constants.

These would be use for Simplicity covenants.

B.1.6.1 11011000100...: Jets for tagged hash IVs

B.1.6.1.1 iden-commitment-tag
B.1.6.1.2 comp-commitment-tag
B.1.6.1.3 unit-commitment-tag
B.1.6.1.4 injl-commitment-tag
B.1.6.1.5 injr-commitment-tag
B.1.6.1.6 case-commitment-tag
B.1.6.1.7 pair-commitment-tag
B.1.6.1.8 take-commitment-tag
B.1.6.1.9 drop-commitment-tag
B.1.6.1.10 witness-commitment-tag
B.1.6.1.11 disconnect-commitment-tag
B.1.6.1.12 fail-commitment-tag
B.1.6.1.13 signtaure-tag

$\ulcorner$'signature-tag'$\urcorner := [110]_2 \cdot \ulcorner 6 \urcorner \cdot \ulcorner 1 \urcorner \cdot \ulcorner 13 \urcorner$
signature-tag: $\mathbb{1} \vdash 2^{256}$

B.1.6.1.14 sighash-tag

B.1.7 110110011...: Jets for Bitcoin (without primitives)

This section is not recommended for non-Bitcoin(-like) applications.

B.1.7.1 parse-lock

(Note: $[\![$'parse-lock'$]\!] \lfloor 0 \rfloor_{32} := \sigma^{\mathbf{L}}(0)$)
$\ulcorner$'parse-lock'$\urcorner := [110]_2 \cdot \ulcorner 7 \urcorner \cdot \ulcorner 1 \urcorner$
parse-lock : $2^{32} \vdash 2^{32} + 2^{32}$

B.1.7.2 parse-sequence

$\ulcorner$'parse-sequence'$\urcorner := [110]_2 \cdot \ulcorner 7 \urcorner \cdot \ulcorner 2 \urcorner$
parse-sequence : $2^{32} \vdash \mathrm{S}(2^{16} + 2^{16})$

B.2 111...: Bitcoin Jets

B.2.1 Signature Hash Modes

- sighash-all:$\mathbb{1} \vdash 2^{256}$
- sighash-single:$\mathbb{1} \vdash 2^{256}$ (anyone can pay)

- sighash-outputs:$\mathrm{Index} \times 2^{32} \vdash 2^{256}$ (anyone can pay)

B.2.2 Time Locks

In this section we define $\mathrm{Height} := 2^{32}$, $\mathrm{Time} := 2^{32}$, $\mathrm{Distance} := 2^{16}$, and $\mathrm{Duration} = 2^{16}$.

B.2.2.1 check-lock-height

$\ulcorner$'check-lock-height'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 1 \urcorner$
check-lock-height : $\mathrm{Height} \vdash \mathbb{1}$

Asserts that the value returned by 'tx-lock-height' is greater than or equal to the input.

B.2.2.2 check-lock-time

$\ulcorner$'check-lock-time'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 2 \urcorner$
'check-lock-time' : $\mathrm{Time} \vdash \mathbb{1}$

Asserts that the value returned by 'tx-lock-time' is greater than or equal to the input.

B.2.2.3 check-lock-distance

$\ulcorner$'check-lock-distance'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 3 \urcorner$
'check-lock-distance' : $\mathrm{Distance} \vdash \mathbb{1}$

Asserts that the value returned by 'check-lock-distance' is greater than or equal to the input.

B.2.2.4 check-lock-duration

$\ulcorner$'check-lock-duration'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 4 \urcorner$
'check-lock-duration' : $\mathrm{Duration} \vdash \mathbb{1}$

Asserts that the value returned by 'check-lock-duration' is greater than or equal to the input.

B.2.2.5 tx-height-lock

$\ulcorner$'tx-height-lock'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 5 \urcorner$
tx-height-lock : $\mathbb{1} \vdash \mathrm{Height}$

Returns x when is-final returns 0_2 and $[\![\text{parse-lock}]\!] \circ [\![\text{lockTime}]\!]$ returns $\sigma^{\mathbf{L}}(x)$.
Otherwise returns $\lfloor 0 \rfloor_{32}$.

B.2.2.6 tx-time-lock

$\ulcorner$'tx-time-lock'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 6 \urcorner$
'tx-time-lock' : $\mathbb{1} \vdash \mathrm{Time}$

Returns x when is-final returns 0_2 and $[\![\text{parse-lock}]\!] \circ [\![\text{lockTime}]\!]$ returns $\sigma^{\mathbf{R}}(x)$.
Otherwise returns $\lfloor 0 \rfloor_{32}$.

B.2.2.7 tx-distance-lock

$\ulcorner$'tx-distance-lock'$\urcorner := [111]_{2}{\cdot}\ulcorner 2\urcorner{\cdot}\ulcorner 7\urcorner$
'tx-distance-lock' : $\mathbb{1} \vdash$ Distance

When $2 \leq \lceil [\![\mathsf{version}]\!]\langle\rangle \rceil_{2^{32}}$, returns the greatest x of values $\eta^{S}(\sigma^{\mathbf{L}}(x))$ returned by $(\mathrm{S}[\![\mathsf{parse\text{-}sequence}]\!] \circ [\![\mathsf{inputSequence}]\!])(i)$ for any input i, if there are any. Otherwise returns $\lfloor 0 \rfloor_{16}$.

B.2.2.8 tx-duration-lock

$\ulcorner$'tx-duration-lock'$\urcorner := [111]_{2}{\cdot}\ulcorner 2\urcorner{\cdot}\ulcorner 8\urcorner$
'tx-duration-lock' : $\mathbb{1} \vdash$ Duration

When $2 \leq \lceil [\![\mathsf{version}]\!]\langle\rangle \rceil_{2^{32}}$, returns the greatest x of values $\eta^{\mathrm{S}}(\sigma^{\mathbf{R}}(x))$ returned by $(\mathrm{S}[\![\mathsf{parse\text{-}sequence}]\!] \circ [\![\mathsf{inputSequence}]\!])(i)$ for any input i, if there are any. Otherwise returns $\lfloor 0 \rfloor_{16}$.

B.2.2.9 is-final

$\ulcorner$'is-final'$\urcorner := [111]_{2}{\cdot}\ulcorner 2\urcorner{\cdot}\ulcorner 9\urcorner$
'is-final' : $\mathbb{1} \vdash \mathbb{2}$

Returns $1_{\mathbb{2}}$ when all the sequence numbers of the tranaction inputs are at their maximum value. Otherwise returns $0_{\mathbb{2}}$.

B.2.3 Transaction

B.2.3.1 script-cmr

$\ulcorner$'script-cmr'$\urcorner := [110]_{2}{\cdot}\ulcorner 3\urcorner{\cdot}\ulcorner 1\urcorner$
script-cmr : $\mathbb{1} \vdash 2^{256}$

B.2.3.2 internal-key

$\ulcorner$'internal-key'$\urcorner := [110]_{2}{\cdot}\ulcorner 3\urcorner{\cdot}\ulcorner 2\urcorner$
internal-key : $\mathbb{1} \vdash$ PubKey

B.2.3.3 current-index

$\ulcorner$'current-index'$\urcorner := [110]_{2}{\cdot}\ulcorner 3\urcorner{\cdot}\ulcorner 3\urcorner$
current-index : $\mathbb{1} \vdash 2^{32}$

B.2.3.4 num-inputs

$\ulcorner$'num-inputs'$\urcorner := [110]_{2}{\cdot}\ulcorner 3\urcorner{\cdot}\ulcorner 4\urcorner$
num-inputs : $\mathbb{1} \vdash 2^{32}$

B.2.3.5 num-outputs

$\ulcorner$'num-outputs'$\urcorner := [110]_{2}{\cdot}\ulcorner 3\urcorner{\cdot}\ulcorner 5\urcorner$
num-outputs : $\mathbb{1} \vdash 2^{32}$

B.2.3.6 lock-time

⌜'lock-time'⌝ := [110]$_2$·⌜3⌝·⌜6⌝
lock-time : $\mathbb{1} \vdash \mathrm{Lock}$

B.2.3.7 fee

⌜'fee'⌝ := [110]$_2$·⌜3⌝·⌜7⌝
fee : $\mathbb{1} \vdash 2^{64}$

B.2.3.8 output-value

⌜'output-value'⌝ := [110]$_2$·⌜3⌝·⌜8⌝
output-value : $2^{32} \vdash \mathrm{S}(2^{64})$

B.2.3.9 output-script-hash

⌜'output-script-hash'⌝ := [110]$_2$·⌜3⌝·⌜9⌝
output-script-hash : $2^{32} \vdash \mathrm{S}(2^{256})$

B.2.3.10 total-output-value

⌜'total-output-value'⌝ := [110]$_2$·⌜3⌝·⌜10⌝
total-output-value : $\mathbb{1} \vdash 2^{64}$

B.2.3.11 current-prev-outpoint

⌜'current-prev-outpoint'⌝ := [110]$_2$·⌜3⌝·⌜11⌝
current-prev-outpoint : $\mathbb{1} \vdash \mathrm{Outpoint}$

B.2.3.12 current-value

⌜'current-value'⌝ := [110]$_2$·⌜3⌝·⌜12⌝
current-value : $\mathbb{1} \vdash 2^{64}$

B.2.3.13 current-script-hash

⌜'current-script-hash'⌝ := [110]$_2$·⌜3⌝·⌜13⌝
current-script-hash : $\mathbb{1} \vdash 2^{256}$

B.2.3.14 current-sequence

⌜'current-sequence'⌝ := [110]$_2$·⌜3⌝·⌜14⌝
current-sequence : $\mathbb{1} \vdash 2^{32}$

B.2.3.15 current-annex-hash

⌜'current-annex-hash'⌝ := [110]$_2$·⌜3⌝·⌜15⌝
current-annex-hash : $\mathbb{1} \vdash \mathrm{S}(2^{256})$

B.2.3.16 current-script-sig-hash

⌜'current-script-sig-hash'⌝ := [110]$_2$·⌜3⌝·⌜16⌝

current-script-sig-hash : $\mathbb{1} \vdash 2^{256}$

B.2.3.17 input-prev-outpoint

$\ulcorner$'input-prev-outpoint'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 17 \urcorner$
input-prev-outpoint : $2^{32} \vdash \mathrm{S}(\mathrm{Outpoint})$

B.2.3.18 input-value

$\ulcorner$'input-value'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 18 \urcorner$
input-value : $2^{32} \vdash \mathrm{S}(2^{64})$

B.2.3.19 input-script-hash

$\ulcorner$'input-script-hash'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 19 \urcorner$
input-script-hash : $2^{32} \vdash \mathrm{S}(2^{256})$

B.2.3.20 input-sequence

$\ulcorner$'input-sequence'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 20 \urcorner$
input-sequence : $2^{32} \vdash \mathrm{S}(2^{32})$

B.2.3.21 input-annex-hash

$\ulcorner$'input-annex-hash'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 21 \urcorner$
input-annex-hash : $2^{32} \vdash \mathrm{S}(\mathrm{S}(2^{256}))$

B.2.3.22 input-script-sig-hash

$\ulcorner$'input-script-sig-hash'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 22 \urcorner$
input-script-sig-hash : $2^{32} \vdash \mathrm{S}(2^{256})$

B.2.3.23 total-input-value

$\ulcorner$'total-input-value'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 23 \urcorner$
total-input-value : $\mathbb{1} \vdash 2^{64}$

B.2.3.24 tapleaf-version

$\ulcorner$'tapleaf-version'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 24 \urcorner$
tapleaf-version : $\mathbb{1} \vdash 2^{8}$

B.2.3.25 tappath

$\ulcorner$'tappath'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 25 \urcorner$
tappath : $2^{8} \vdash \mathrm{S}(2^{256})$

B.2.3.26 version

$\ulcorner$'version'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 26 \urcorner$
version : $\mathbb{1} \vdash 2^{32}$

B.3 111...: Elements Jets

B.3.1 Signature Hash Modes

B.3.1.1 sig-all-hash

$\ulcorner$'sig-all-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 1 \urcorner$
sig-all-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the following:

- The result of genesis-block-hash (32 bytes).
- The result of genesis-block-hash again (32 bytes).
- The result of tx-hash (32 bytes).
- The result of tap-env-hash (32 bytes).
- The result of current-index (Note: this is in big endian format) (4 bytes).

Note: the two copies of the genesis-block-hash values effectively makes this result a BIP-340 style tagged hash.

B.3.1.2 tx-hash

$\ulcorner$'tx-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 2 \urcorner$
tx-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the following:

- The result of version (Note: this is in big endian format) (4 bytes).
- The result of lock (Note: this is in big endian format) (4 bytes).
- The result of inputs-hash (32 bytes).
- The result of outputs-hash (32 bytes).
- The result of issuances-hash (32 bytes).
- The result of output-surjection-proofs-hash (32 bytes).
- The result of input-utxos-hash (32 bytes).

B.3.1.3 tap-env-hash

$\ulcorner$'tap-env-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 3 \urcorner$
tap-env-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the following:

- The result of tapleaf-hash (32 bytes).
- The result of tappath-hash (32 bytes).
- The result of internal-key (32 bytes).

B.3.1.4 inputs-hash

$\ulcorner$'inputs-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 4 \urcorner$
inputs-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the following:

- The result of input-outpoints-hash (32 bytes).
- The result of input-sequences-hash (32 bytes).
- The result of input-annexes-hash (32 bytes).

B.3.1.5 outputs-hash

$\ulcorner$'outputs-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 5 \urcorner$
outputs-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the following:

- The result of output-amounts-hash (32 bytes).
- The result of output-nonces-hash (32 bytes).
- The result of output-scripts-hash (32 bytes).
- The result of output-range-proofs-hash (32 bytes).

Note: the result of output-surjection-proofs-hash is specifically excluded because surjection proofs are dependent on the inputs as well as the output. See also tx-hash.

B.3.1.6 issuances-hash

$\ulcorner$'issuances-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 6 \urcorner$
issuances-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the following:

- The result of issuance-asset-amounts-hash (32 bytes).
- The result of issuance-token-amounts-hash (32 bytes).
- The result of issuance-range-proofs-hash (32 bytes).
- The result of issuance-blinding-entropy-hash (32 bytes).

B.3.1.7 input-utxos-hash

$\ulcorner$'input-utxos-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 7 \urcorner$
input-utxos-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the following:

- The result of input-amounts-hash (32 bytes).
- The result of input-scripts-hash (32 bytes).

B.3.1.8 output-amounts-hash

$\ulcorner$'output-amounts-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 8 \urcorner$

output-amounts-hash : $\mathbb{1} \vdash 2^{256}$

Returns the SHA256 hash of the serialization of each output's asset and amount fields.

B.3.1.9 output-scripts-hash

$\ulcorner$'output-scripts-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 9 \urcorner$
output-scripts-hash : $\mathbb{1} \vdash 2^{256}$

Returns the SHA256 hash of the concatenation of the SHA256 hash of each output's scriptPubKey.

B.3.1.10 output-nonces-hash

$\ulcorner$'output-nonces-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 10 \urcorner$
output-nonces-hash : $\mathbb{1} \vdash 2^{256}$

Returns the SHA256 hash of the serialization of each output's nonce field.

B.3.1.11 output-range-proofs-hash

$\ulcorner$'output-range-proofs-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 11 \urcorner$
output-range-proofs-hash : $\mathbb{1} \vdash 2^{256}$

Returns the SHA256 hash of the concatenation of the SHA256 hash of each output's range proof.

Note that if the output's amount is explicit then the range proof is considered the empty string.

B.3.1.12 output-surjection-proofs-hash

$\ulcorner$'output-surjection-proofs-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 12 \urcorner$
output-surjection-proofs-hash : $\mathbb{1} \vdash 2^{256}$

Returns the SHA256 hash of the concatenation of the SHA256 hash of each output's surjection proof.

Note that if the output's asset is explicit then the surjection proof is considered the empty string.

B.3.1.13 input-outpoints-hash

$\ulcorner$'input-outpoints-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 13 \urcorner$
input-outpoints-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the concatenation of the following for every input.

- If the input is not a pegin, then the byte `0x00`.
- If the input is a pegin, then the byte `0x01` followed by the parent chain's genesis hash (32 bytes).
- The input's serialized previous transaction id (32 bytes).
- The input's previous transaction index in big endian format (4 bytes).

IMPORTANT: the index is serialized in big endian format rather than little endian format.

B.3.1.14 input-sequences-hash

$\ulcorner$'input-sequences-hash'$\urcorner := [111]_{2^8}\cdot\ulcorner 1\urcorner\cdot\ulcorner 14\urcorner$
input-sequences-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the concatenation of the following for every input.

- The inputs sequence number in big endian format (4 bytes).

IMPORTANT, the sequence number is serialized in big endian format rather than little endian format.

B.3.1.15 input-annexes-hash

$\ulcorner$'input-annexes-hash'$\urcorner := [111]_{2^8}\cdot\ulcorner 1\urcorner\cdot\ulcorner 15\urcorner$
input-annexes-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the concatenation of the following for every input.

- If the input has no annex, or isn't a taproot spend, then the byte `0x00`.
- If the input has an annex, then the byte `0x01` followed by a SHA256 hash of the annex (32 bytes).

B.3.1.16 input-script-sigs-hash

$\ulcorner$'input-script-sigs-hash'$\urcorner := [111]_{2^8}\cdot\ulcorner 1\urcorner\cdot\ulcorner 16\urcorner$
input-script-sigs-hash : $\mathbb{1} \vdash 2^{256}$

Returns the SHA256 hash of the concatenation of the SHA256 hash of each input's scriptSig.

Note that if an input's UTXO uses segwit, then it's scriptSig will necessarily be the empty string. In such cases we still use the SHA256 hash of the empty string.

B.3.1.17 issuance-asset-amounts-hash

$\ulcorner$'issuance-asset-amounts-hash'$\urcorner := [111]_{2^8}\cdot\ulcorner 1\urcorner\cdot\ulcorner 17\urcorner$
issuance-asset-amounts-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the concatenation of the following for every input.

- If the input has no issuance then two bytes `0x00 0x00`.
- If the input is has a new issuance then the byte `0x01` followed by a serialization of the calculated issued asset id (32 bytes) followed by the serialization of the (possibly confidential) issued asset amount (9 bytes or 33 bytes).
- If the input is has a reissuance then the byte `0x01` followed by a serialization of the issued asset id (32 bytes), followed by the serialization of the (possibly confidential) issued asset amount (9 bytes or 33 bytes).

IMPORTANT: If there is an issuance but there are no asset issued (i.e. the amount is null) we serialize the vase as the explicit 0 amount, (i.e. `0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00`).

Note, the issuance asset id is serialized in the same format as an explicit asset id would be.

B.3.1.18 issuance-token-amounts-hash

$\ulcorner$'issuance-token-amounts-hash'$\urcorner := [111]_2.\ulcorner 1 \urcorner.\ulcorner 18 \urcorner$
issuance-token-amounts-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the concatenation of the following for every input.

- If the input has no issuance then two bytes `0x00 0x00`.
- If the input is has a new issuance then the byte `0x01` followed by a serialization of the calculated issued token id (32 bytes) followed by the serialization of the (possibly confidential) issued token amount (9 bytes or 33 bytes).
- If the input is has a reissuance then the byte `0x01` followed by a serialization of the issued token id (32 bytes), followed by the serialization of the explicit 0 amount (i.e `0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00`) (9 bytes).

IMPORTANT: If there is an issuance but there are no tokens issued (i.e. the amount is null) we serialize the vase as the explicit 0 amount, (i.e. `0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00`).

Note, the issuance token id is serialized in the same format as an explicit asset id would be.

B.3.1.19 issuance-range-proofs-hash

$\ulcorner$'issuance-range-proofs-hash'$\urcorner := [111]_2.\ulcorner 1 \urcorner.\ulcorner 19 \urcorner$
issuance-range-proofs-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the concatenation of the following for every input.

- The SHA256 hash of the range proof of the input's issuance asset amount (32 bytes).
- The SHA256 hash of the range proof of the input's issuance token amount (32 bytes).

Note that each the range proof is considered to be the empty string in the case there is no issuance, or if the asset or token amount doesn't exist (i.e is null). The SHA256 hash of the empty string is still used in these cases.

B.3.1.20 issuance-blinding-entropy-hash

$\ulcorner$'issuance-blinding-entropy-hash'$\urcorner := [111]_2.\ulcorner 1 \urcorner.\ulcorner 20 \urcorner$
issuance-blinding-entropy-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the concatenation of the following for every input.

- If the input has no issuance then the byte `0x00`.
- If the input is has a new issuance then the byte `0x01` followed by 32 `0x00` bytes and the new issuance's contract hash field (32 bytes).
- If the input is has reissuance then the byte `0x01` followed by a serializaiton of the reissuance's blinding nonce field (32 bytes) and the reissuance's entropy field (32 bytes).

Note that if the issuance is a new issuance then the blinding nonce field is 32 `0x00` bytes and new issuance's contract hash

B.3.1.21 input-amounts-hash

$\ulcorner$'input-amounts-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 21 \urcorner$
input-amounts-hash : $\mathbb{1} \vdash 2^{256}$

Returns the SHA256 hash of the serialization of each input UTXO's asset and amount fields.

B.3.1.22 input-scripts-hash

$\ulcorner$'input-scripts-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 22 \urcorner$
input-scripts-hash : $\mathbb{1} \vdash 2^{256}$

Returns the SHA256 hash of the concatenation of the SHA256 hash of each input UTXO's scriptPubKey.

B.3.1.23 tapleaf-hash

$\ulcorner$'tapleaf-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 23 \urcorner$
tapleaf-hash : $\mathbb{1} \vdash 2^{256}$

Returns a SHA256 hash of the following:

- The hash of the ASCII string `TapLeaf/elements` (32 bytes).
- The hash of the ASCII string `TapLeaf/elements` again (32 bytes).
- The result of tapleaf-version (1 byte).
- The byte `0x20` (1 byte).
- The result of scriptCMR (32 bytes).

Note: this matches Element's modified BIP-0341 definition of tapleaf hash.

B.3.1.24 tappath-hash

$\ulcorner$'tappath-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 24 \urcorner$
tappath-hash : $\mathbb{1} \vdash 2^{256}$

Returns a hash of the current input's control block excluding the leaf version and the taproot internal key. Using the notation of BIP-0341, it returns the SHA256 hash of $c[33{:}\,33 + 32m]$.

B.3.1.25 outpoint-hash

$\ulcorner$'outpoint-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 25 \urcorner$
outpoint-hash : $\mathrm{Ctx8} \times (\mathrm{S}(2^{256}) \times \mathrm{Outpoint}) \vdash \mathrm{Ctx8}$

Continues a SHA256 hash with an optional pegin and an outpoint by appending the following:

- If the input is not a pegin, then the byte `0x00`.
- If the input is a pegin, then the byte `0x01` followed by the given parent genesis hash (32 bytes).
- The input's previous transaction id (32 bytes).
- The input's previous transaction index in big endian format (4 bytes).

B.3.1.26 asset-amount-hash

$\ulcorner$'asset-amount-hash'$\urcorner := [111]_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 26 \urcorner$

asset-amount-hash : $\mathrm{Ctx8} \times (\mathrm{Asset}(\mathbb{1}) \times \mathrm{Amount}(\mathbb{1})) \vdash \mathrm{Ctx8}$

Continues a SHA256 hash with the serialization of a confidential asset followed by the serialization of a confidential amount.

B.3.1.27 nonce-hash

$\ulcorner$'nonce-hash'$\urcorner := \mathtt{[111]}_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 27 \urcorner$
nonce-hash : $\mathrm{Ctx8} \times (\mathrm{S}(\mathrm{Nonce})) \vdash \mathrm{Ctx8}$

Continues a SHA256 hash with the serialization of an optional nonce.

B.3.1.28 annex-hash

$\ulcorner$'annex-hash'$\urcorner := \mathtt{[111]}_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 28 \urcorner$
annex-hash : $\mathrm{Ctx8} \times (\mathrm{S}(2^{256})) \vdash \mathrm{Ctx8}$

Continues a SHA256 hash with an optional hash by appending the following:

- If there is no hash, then the byte `0x00`.
- If there is a hash, then the byte `0x01` followed by the given hash (32 bytes).

B.3.1.29 build-tapleaf-simplicity

$\ulcorner$'build-tapleaf-simplicity'$\urcorner := \mathtt{[111]}_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 29 \urcorner$
build-tapleaf-simplicity : $2^{256} \vdash 2^{256}$

Returns a SHA256 hash of the following:

- The hash of the ASCII string `TapLeaf/elements` (32 bytes).
- The hash of the ASCII string `TapLeaf/elements` again (32 bytes).
- The byte `0xbe` (1 byte) for the Simplicity tapleaf version number.
- The byte `0x20` (1 byte).
- The input (32 bytes) which should be the CMR of a Simplicity program.

B.3.1.30 build-tapbranch

$\ulcorner$'build-tapbranch-simplicity'$\urcorner := \mathtt{[111]}_2 \cdot \ulcorner 1 \urcorner \cdot \ulcorner 30 \urcorner$
build-tapbranch-simplicity : $2^{256} \times 2^{256} \vdash 2^{256}$

Returns a SHA256 hash of the following:

- The hash of the ASCII string `TapBranch/elements` (32 bytes).
- The hash of the ASCII string `TapBranch/elements` again (32 bytes).
- The lexicographically smaller of the two inputs (32 bytes).
- The lexicographically larger of the two inputs (32 bytes).

This builds a taproot from two branches.

B.3.2 Time Locks

In this section we define $\mathrm{Height} := 2^{32}$, $\mathrm{Time} := 2^{32}$, $\mathrm{Distance} := 2^{16}$, and $\mathrm{Duration} = 2^{16}$.

B.3.2.1 check-lock-height

$\ulcorner$'check-lock-height'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 1 \urcorner$
check-lock-height : Height $\vdash \mathbb{1}$

Asserts that the value returned by 'tx-lock-height' is greater than or equal to the input.

B.3.2.2 check-lock-time

$\ulcorner$'check-lock-time'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 2 \urcorner$
'check-lock-time' : Time $\vdash \mathbb{1}$

Asserts that the value returned by 'tx-lock-time' is greater than or equal to the input.

B.3.2.3 check-lock-distance

$\ulcorner$'check-lock-distance'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 3 \urcorner$
'check-lock-distance' : Distance $\vdash \mathbb{1}$

Asserts that the value returned by 'check-lock-distance' is greater than or equal to the input.

B.3.2.4 check-lock-duration

$\ulcorner$'check-lock-duration'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 4 \urcorner$
'check-lock-duration' : Duration $\vdash \mathbb{1}$

Asserts that the value returned by 'check-lock-duration' is greater than or equal to the input.

B.3.2.5 tx-lock-height

$\ulcorner$'tx-lock-height'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 5 \urcorner$
tx-lock-height : $\mathbb{1} \vdash$ Height

Returns x when is-final returns 0_2 and $[\![\text{parse-lock}]\!] \circ [\![\text{lockTime}]\!]$ returns $\sigma^{\mathbf{L}}(x)$.
Otherwise returns $\lfloor 0 \rfloor_{32}$.

B.3.2.6 tx-lock-time

$\ulcorner$'tx-lock-time'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 6 \urcorner$
'tx-lock-time' : $\mathbb{1} \vdash$ Time

Returns x when is-final returns 0_2 and $[\![\text{parse-lock}]\!] \circ [\![\text{lockTime}]\!]$ returns $\sigma^{\mathbf{R}}(x)$.
Otherwise returns $\lfloor 0 \rfloor_{32}$.

B.3.2.7 tx-lock-distance

$\ulcorner$'tx-lock-distance'$\urcorner := [111]_2 \cdot \ulcorner 2 \urcorner \cdot \ulcorner 7 \urcorner$
'tx-lock-distance' : $\mathbb{1} \vdash$ Distance

When $2 \leq \lceil [\![\text{version}]\!]\langle\rangle \rceil_{2^{32}}$, returns the greatest x of values $\eta^S(\sigma^{\mathbf{L}}(x))$ returned by $(\mathrm{S}[\![\text{parse-sequence}]\!] \circ [\![\text{inputSequence}]\!])(i)$ for any input i, if there are any. Otherwise returns $\lfloor 0 \rfloor_{16}$.

B.3.2.8 tx-lock-duration

⌜'tx-lock-duration'⌝ := [111]$_2$·⌜2⌝·⌜8⌝
'tx-lock-duration' : $\mathbb{1} \vdash$ Duration

When $2 \leq \lceil [\![\mathsf{version}]\!]\langle\rangle \rceil_{2^{32}}$, returns the greatest x of values $\eta^{\mathrm{S}}(\sigma^{\mathbf{R}}(x))$ returned by $(\mathrm{S}[\![\mathsf{parse\text{-}sequence}]\!] \circ [\![\mathsf{inputSequence}]\!])(i)$ for any input i, if there are any. Otherwise returns $\lfloor 0 \rfloor_{16}$.

B.3.2.9 tx-is-final

⌜'tx-is-final'⌝ := [111]$_2$·⌜2⌝·⌜9⌝
'tx-is-final' : $\mathbb{1} \vdash \mathbb{2}$

Returns 1_2 when all the sequence numbers of the tranaction inputs are at their maximum value.
Otherwise returns 0_2.

B.3.3 Issuance

B.3.3.1 issuance

⌜'issuance'⌝ := [110]$_2$·⌜3⌝·⌜1⌝
issuance : Index $\vdash \mathrm{S}(\mathrm{S}(\mathbb{2}))$

Returns $\eta(\eta(0_2))$ when the given input has a new issuance. Returns $\eta(\eta(1_2))$ when the given input has a reissuance. Returns $\eta(\emptyset)$ when the given input has no issuance.
Otherwise returns $\emptyset$ when there is no input of the given index.

B.3.3.2 issuance-asset

⌜'issuance-asset'⌝ := [110]$_2$·⌜3⌝·⌜2⌝
issuance-asset : Index $\vdash \mathrm{S}(\mathrm{S}(\mathrm{ExplicitAsset}))$

Returns $\eta(\eta(x))$ when the given input has an issuance with asset ID x. Returns $\eta(\emptyset)$ when the given input has no issuance.
Otherwise returns $\emptyset$ when there is no input of the given index.

B.3.3.3 issuance-token

⌜'issuance-token'⌝ := [110]$_2$·⌜3⌝·⌜3⌝
issuance-token : Index $\vdash \mathrm{S}(\mathrm{S}(\mathrm{ExplicitAsset}))$

Returns $\eta(\eta(x))$ when the given input has an issuance with reissuance token ID x. Returns $\eta(\emptyset)$ when the given input has no issuance.
Otherwise returns $\emptyset$ when there is no input of the given index.

B.3.3.4 issuance-entropy

⌜'issuance-token'⌝ := [110]$_2$·⌜3⌝·⌜4⌝
issuance-token : Index $\vdash \mathrm{S}(\mathrm{S}(2^{256}))$

Returns $\eta(\eta(x))$ when the given input has an reissuance with entropy x, or if there is a new issuance whose computed entropy is x. Returns $\eta(\emptyset)$ when the given input has no issuance.

Otherwise returns $\emptyset$ when there is no input of the given index.

B.3.3.5 calculate-issuance-entropy

$\ulcorner$'calculate-issuance-entropy'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 5 \urcorner$
calculate-issuance-entropy : $\text{Outpoint} \times 2^{256} \vdash 2^{256}$

Calculate the entropy value from a given ouptoint and contract hash. This entropy value is used to compute issued asset and token IDs (see below).

B.3.3.6 calculate-asset

$\ulcorner$'calculate-asset'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 6 \urcorner$
calculate-asset : $2^{256} \vdash \text{ExplicitAsset}$

Calculate the issued asset id from a given entropy value.

B.3.3.7 calculate-explicit-token

$\ulcorner$'calculate-token'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 7 \urcorner$
calculate-token : $2^{256} \vdash \text{ExplicitAsset}$

Calculate the reissuance token id from a given entropy value for assets with explicit issued amounts.

B.3.3.8 calculate-confidential-token

$\ulcorner$'calculate-token'$\urcorner := [110]_2 \cdot \ulcorner 3 \urcorner \cdot \ulcorner 8 \urcorner$
calculate-token : $2^{256} \vdash \text{ExplicitAsset}$

Calculate the reissuance token id from a given entropy value for assets with confidential issued amounts.

B.3.4 Transaction

B.3.4.1 script-cmr

$\ulcorner$'script-cmr'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 1 \urcorner$
script-cmr : $\mathbb{1} \vdash 2^{256}$

B.3.4.2 internal-key

$\ulcorner$'internal-key'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 2 \urcorner$
internal-key : $\mathbb{1} \vdash \text{PubKey}$

B.3.4.3 current-index

$\ulcorner$'current-index'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 3 \urcorner$
current-index : $\mathbb{1} \vdash 2^{32}$

B.3.4.4 num-inputs

$\ulcorner$'num-inputs'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 4 \urcorner$
num-inputs : $\mathbb{1} \vdash 2^{32}$

B.3.4.5 num-outputs

⌜'num-outputs'⌝ := [110]$_2$·⌜4⌝·⌜5⌝
num-outputs : $\mathbb{1} \vdash 2^{32}$

B.3.4.6 lock-time

⌜'lock-time'⌝ := [110]$_2$·⌜4⌝·⌜6⌝
lock-time : $\mathbb{1} \vdash \mathrm{Lock}$

B.3.4.7 output-asset

⌜'output-asset'⌝ := [110]$_2$·⌜4⌝·⌜7⌝
output-asset : $2^{32} \vdash \mathrm{S}(\mathrm{Asset}(\mathbb{1}))$

B.3.4.8 output-amount

⌜'output-amount'⌝ := [110]$_2$·⌜4⌝·⌜8⌝
output-amount : $2^{32} \vdash \mathrm{S}(\mathrm{Asset}(\mathbb{1}) \times \mathrm{Amount}(\mathbb{1}))$

B.3.4.9 output-nonce

⌜'output-nonce'⌝ := [110]$_2$·⌜4⌝·⌜9⌝
output-nonce : $2^{32} \vdash \mathrm{S}(\mathrm{S}(\mathrm{Nonce}))$

B.3.4.10 output-script-hash

⌜'output-script-hash'⌝ := [110]$_2$·⌜4⌝·⌜10⌝
output-script-hash : $2^{32} \vdash \mathrm{S}(2^{256})$

B.3.4.11 output-null-datum

⌜'output-null-datum'⌝ := [110]$_2$·⌜4⌝·⌜11⌝
output-null-datum : $2^{32} \times 2^{32} \vdash \mathrm{S}(\mathrm{S}(2^2 \times 2^{256} + (2 + 2^4)))$

B.3.4.12 output-is-fee

⌜'output-is-fee'⌝ := [110]$_2$·⌜4⌝·⌜12⌝
output-is-fee : $2^{32} \vdash \mathrm{S}(2)$

B.3.4.13 output-surjection-proof

⌜'output-surjection-proof'⌝ := [110]$_2$·⌜4⌝·⌜13⌝
output-surjection-proof : $2^{32} \vdash \mathrm{S}(2^{256})$

B.3.4.14 output-range-proof

⌜'output-range-proof'⌝ := [110]$_2$·⌜4⌝·⌜14⌝
output-range-proof : $2^{32} \vdash \mathrm{S}(2^{256})$

B.3.4.15 total-fee

$\ulcorner$'total-fee'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 15 \urcorner$
total-fee : $\mathrm{ExplicitAsset} \vdash \mathrm{ExplicitAmount}$

B.3.4.16 current-pegin

$\ulcorner$'current-pegin'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 16 \urcorner$
current-pegin : $\mathbb{1} \vdash \mathrm{S}(2^{256})$

B.3.4.17 current-prev-outpoint

$\ulcorner$'current-prev-outpoint'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 17 \urcorner$
current-prev-outpoint : $\mathbb{1} \vdash \mathrm{Outpoint}$

B.3.4.18 current-asset

$\ulcorner$'current-asset'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 18 \urcorner$
current-asset : $\mathbb{1} \vdash \mathrm{Asset}(\mathbb{1})$

B.3.4.19 current-amount

$\ulcorner$'current-amount'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 19 \urcorner$
current-amount : $\mathbb{1} \vdash \mathrm{Asset}(\mathbb{1}) \times \mathrm{Amount}(\mathbb{1})$

B.3.4.20 current-script-hash

$\ulcorner$'current-script-hash'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 20 \urcorner$
current-script-hash : $\mathbb{1} \vdash 2^{256}$

B.3.4.21 current-sequence

$\ulcorner$'current-sequence'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 21 \urcorner$
current-sequence : $\mathbb{1} \vdash 2^{32}$

B.3.4.22 current-annex-hash

$\ulcorner$'current-annex-hash'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 22 \urcorner$
current-annex-hash : $\mathbb{1} \vdash \mathrm{S}(2^{256})$

B.3.4.23 current-script-sig-hash

$\ulcorner$'current-script-sig-hash'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 23 \urcorner$
current-script-sig-hash : $\mathbb{1} \vdash 2^{256}$

B.3.4.24 current-reissuance-blinding

$\ulcorner$'current-reissuance-blinding'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 24 \urcorner$
current-reissuance-blinding : $\mathbb{1} \vdash \mathrm{S}(\mathrm{ExplicitNonce})$

B.3.4.25 current-new-issuance-contract

$\ulcorner$'current-new-issuance-contract'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 25 \urcorner$
current-new-issuance-contract : $\mathbb{1} \vdash \mathrm{S}(2^{256})$

B.3.4.26 current-reissuance-entropy

$\ulcorner$'current-reissuance-entropy'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 26 \urcorner$
current-reissuance-entropy : $\mathbb{1} \vdash \mathrm{S}(2^{256})$

B.3.4.27 current-issuance-asset-amount

$\ulcorner$'current-issuance-asset-amount'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 27 \urcorner$
current-issuance-asset-amount : $\mathbb{1} \vdash \mathrm{S}(\mathrm{Amount}(\mathbb{1}))$

B.3.4.28 current-issuance-token-amount

$\ulcorner$'current-issuance-token-amount'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 28 \urcorner$
current-issuance-token-amount : $\mathbb{1} \vdash \mathrm{S}(\mathrm{TokenAmount}(\mathbb{1}))$

B.3.4.29 current-issuance-asset-proof

$\ulcorner$'current-issuance-asset-proof'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 29 \urcorner$
current-issuance-asset-proof : $\mathbb{1} \vdash 2^{256}$

B.3.4.30 current-issuance-token-proof

$\ulcorner$'current-issuance-token-proof'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 30 \urcorner$
current-issuance-token-proof : $\mathbb{1} \vdash 2^{256}$

B.3.4.31 input-pegin

$\ulcorner$'input-pegin'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 31 \urcorner$
input-pegin : $2^{32} \vdash \mathrm{S}(\mathrm{S}(2^{256}))$

B.3.4.32 input-prev-outpoint

$\ulcorner$'input-prev-outpoint'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 32 \urcorner$
input-prev-outpoint : $2^{32} \vdash \mathrm{S}(\mathrm{Outpoint})$

B.3.4.33 input-asset

$\ulcorner$'input-asset'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 33 \urcorner$
input-asset : $2^{32} \vdash \mathrm{S}(\mathrm{Asset}(\mathbb{1}))$

B.3.4.34 input-amount

$\ulcorner$'input-amount'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 34 \urcorner$
input-amount : $2^{32} \vdash \mathrm{S}(\mathrm{Asset}(\mathbb{1}) \times \mathrm{Amount}(\mathbb{1}))$

B.3.4.35 input-script-hash

$\ulcorner$'input-script-hash'$\urcorner := [110]_2 \cdot \ulcorner 4 \urcorner \cdot \ulcorner 35 \urcorner$
input-script-hash : $2^{32} \vdash \mathrm{S}(2^{256})$

B.3.4.36 input-sequence

⌜'input-sequence'⌝ := [110]$_2$·⌜4⌝·⌜36⌝
input-sequence : $2^{32} \vdash \mathrm{S}(2^{32})$

B.3.4.37 input-annex-hash

⌜'input-annex-hash'⌝ := [110]$_2$·⌜4⌝·⌜37⌝
input-annex-hash : $2^{32} \vdash \mathrm{S}(\mathrm{S}(2^{256}))$

B.3.4.38 input-script-sig-hash

⌜'input-script-sig-hash'⌝ := [110]$_2$·⌜4⌝·⌜38⌝
input-script-sig-hash : $2^{32} \vdash \mathrm{S}(2^{256})$

B.3.4.39 reissuance-blinding

⌜'reissuance-blinding'⌝ := [110]$_2$·⌜4⌝·⌜39⌝
reissuance-blinding : $2^{32} \vdash \mathrm{S}(\mathrm{S}(\mathrm{ExplicitNonce}))$

B.3.4.40 new-issuance-contract

⌜'new-issuance-contract'⌝ := [110]$_2$·⌜4⌝·⌜40⌝
new-issuance-contract : $2^{32} \vdash \mathrm{S}(\mathrm{S}(2^{256}))$

B.3.4.41 reissuance-entropy

⌜'reissuance-entropy'⌝ := [110]$_2$·⌜4⌝·⌜41⌝
reissuance-entropy : $2^{32} \vdash \mathrm{S}(\mathrm{S}(2^{256}))$

B.3.4.42 issuance-asset-amount

⌜'issuance-asset-amount'⌝ := [110]$_2$·⌜4⌝·⌜42⌝
issuance-asset-amount : $2^{32} \vdash \mathrm{S}(\mathrm{S}(\mathrm{Amount}(\mathbb{1})))$

B.3.4.43 issuance-token-amount

⌜'issuance-token-amount'⌝ := [110]$_2$·⌜4⌝·⌜43⌝
issuance-token-amount : $2^{32} \vdash \mathrm{S}(\mathrm{S}(\mathrm{TokenAmount}(\mathbb{1})))$

B.3.4.44 issuance-asset-proof

⌜'issuance-asset-proof'⌝ := [110]$_2$·⌜4⌝·⌜44⌝
issuance-asset-proof : $2^{32} \vdash \mathrm{S}(2^{256})$

B.3.4.45 issuance-token-proof

⌜'issuance-token-proof'⌝ := [110]$_2$·⌜4⌝·⌜45⌝
issuance-token-proof : $2^{32} \vdash \mathrm{S}(2^{256})$

B.3.4.46 tapleaf-version

⌜'tapleaf-version'⌝ := [110]$_2$·⌜4⌝·⌜46⌝

tapleaf-version : $\mathbb{1} \vdash 2^{8}$

B.3.4.47 tappath

$\ulcorner$'tappath'$\urcorner := [110]_{2} \cdot \ulcorner 4 \urcorner \cdot \ulcorner 47 \urcorner$
tappath : $2^{8} \vdash \mathrm{S}(2^{256})$

B.3.4.48 version

$\ulcorner$'version'$\urcorner := [110]_{2} \cdot \ulcorner 4 \urcorner \cdot \ulcorner 48 \urcorner$
version : $\mathbb{1} \vdash 2^{32}$

B.3.4.49 genesis-block-hash

$\ulcorner$'genesis-block-hash'$\urcorner := [110]_{2} \cdot \ulcorner 4 \urcorner \cdot \ulcorner 49 \urcorner$

genesis-block-hash : $\mathbb{1} \vdash \mathbb{2}^{256}$

Appendix C
Alternative Serialization of Simplicity DAGs

DEPRICATED

This appendix presents an alternative, byte-oriented prefix code for Simplicity DAGs. This code is not as compact as the bit-oriented code presented in Section 7.2 and it imposes some arbitrary limits on the size of the DAG and witness values. Its only advantage is that it might be faster to decode. This code probably should not be used by any application and maybe should be removed from this report.

First we define a byte-string serialization for numbers i known to be less than a bound b by determining the minimum number of bytes needed to fit a number less than b.

$$\begin{aligned}
\text{byteCode}(1, 0) &:= \epsilon_{2^8} \\
\text{byteCode}(b, i) &:= \text{byteCode}(2^{8n}, q)\cdot\text{BE}(\lfloor i \rfloor_8) \qquad \text{where } 2^8\, q \le i < 2^8(q+1) \text{ and } 2^{8n} < b \le 2^{8(n+1)} \text{ and } i < b
\end{aligned}$$

For witness data, which is a bit-string, we group the bits into bytes to form a byte-string, padding the least significant bits with zero bits.

$$\begin{aligned}
\text{bytePad}(\epsilon) &:= \epsilon_{2^8} \\
\text{bytePad}(b_0 \blacktriangleleft b_1 \blacktriangleleft b_2 \blacktriangleleft b_3 \blacktriangleleft b_4 \blacktriangleleft b_5 \blacktriangleleft b_6 \blacktriangleleft b_7 \blacktriangleleft v) &:= \langle\langle\langle b_0, b_1\rangle, \langle b_2, b_3\rangle\rangle, \langle\langle b_4, b_5\rangle, \langle b_6, b_7\rangle\rangle\rangle \blacktriangleleft \text{bytePad}(v) \\
\text{bytePad}(v) &:= \text{bytePad}(v\cdot\texttt{[0]}^{8-|v|}) \qquad \text{when } 0 < |v| < 8
\end{aligned}$$

Note that by itself bytePad is not a prefix code and forgets how many bits where in the vector. Both issues are addressed by prefixing this encoding with the length of the number of bits in the original bit-string.

Next we define a byte-string serialization for Node, $\text{byteCode} : \mathbb{N} \times \text{Node} \to (2^8)^*$. The first parameter to byteCode, k, is the index at which the Node occurs in the Dag. It is used to determine the bound on the size of the sub-expression offsets for serialization of those values.

$$\begin{aligned}
\text{byteCode}(k, \text{`comp'}\, 1\, 1) &= \texttt{[00]}_{2^8} \\
\text{byteCode}(k, \text{`case'}\, 1\, 1) &= \texttt{[01]}_{2^8} \\
\text{byteCode}(k, \text{`pair'}\, 1\, 1) &= \texttt{[02]}_{2^8} \\
\text{byteCode}(k, \text{`disconnect'}\, 1\, 1) &= \texttt{[03]}_{2^8} \\
\text{byteCode}(k, \text{`comp'}\, 1\, j) &= \texttt{[04]}_{2^8}\cdot\text{byteCode}(k-1, j-2) && \text{where } 1 < j \\
\text{byteCode}(k, \text{`case'}\, 1\, j) &= \texttt{[05]}_{2^8}\cdot\text{byteCode}(k-1, j-2) && \text{where } 1 < j \\
\text{byteCode}(k, \text{`pair'}\, 1\, j) &= \texttt{[06]}_{2^8}\cdot\text{byteCode}(k-1, j-2) && \text{where } 1 < j \\
\text{byteCode}(k, \text{`disconnect'}\, 1\, j) &= \texttt{[07]}_{2^8}\cdot\text{byteCode}(k-1, j-2) && \text{where } 1 < j \\
\text{byteCode}(k, \text{`comp'}\, i\, 1) &= \texttt{[08]}_{2^8}\cdot\text{byteCode}(k-1, i-2) && \text{where } 1 < i \\
\text{byteCode}(k, \text{`case'}\, i\, 1) &= \texttt{[09]}_{2^8}\cdot\text{byteCode}(k-1, i-2) && \text{where } 1 < i \\
\text{byteCode}(k, \text{`pair'}\, i\, 1) &= \texttt{[0a]}_{2^8}\cdot\text{byteCode}(k-1, i-2) && \text{where } 1 < i \\
\text{byteCode}(k, \text{`disconnect'}\, i\, 1) &= \texttt{[0b]}_{2^8}\cdot\text{byteCode}(k-1, i-2) && \text{where } 1 < i \\
\text{byteCode}(k, \text{`comp'}\, i\, j) &= \texttt{[0c]}_{2^8}\cdot\text{byteCode}(k-1, i-2)\cdot\text{byteCode}(k-1, j-2) && \text{where } 1 < i \text{ and } 1 < j \\
\text{byteCode}(k, \text{`case'}\, i\, j) &= \texttt{[0d]}_{2^8}\cdot\text{byteCode}(k-1, i-2)\cdot\text{byteCode}(k-1, j-2) && \text{where } 1 < i \text{ and } 1 < j \\
\text{byteCode}(k, \text{`pair'}\, i\, j) &= \texttt{[0e]}_{2^8}\cdot\text{byteCode}(k-1, i-2)\cdot\text{byteCode}(k-1, j-2) && \text{where } 1 < i \text{ and } 1 < j \\
\text{byteCode}(k, \text{`disconnect'}\, i\, j) &= \texttt{[0f]}_{2^8}\cdot\text{byteCode}(k-1, i-2)\cdot\text{byteCode}(k-1, j-2) && \text{where } 1 < i \text{ and } 1 < j
\end{aligned}$$

$$\begin{aligned}
\text{byteCode}(k, \text{`injl' } 1) &= \texttt{[10]}_{2^8} \\
\text{byteCode}(k, \text{`injr' } 1) &= \texttt{[11]}_{2^8} \\
\text{byteCode}(k, \text{`take' } 1) &= \texttt{[12]}_{2^8} \\
\text{byteCode}(k, \text{`drop' } 1) &= \texttt{[13]}_{2^8} \\
\text{byteCode}(k, \text{`injl' } i) &= \texttt{[18]}_{2^8} \cdot \text{byteCode}(k-1, i-2) && \text{where } 1 < i \\
\text{byteCode}(k, \text{`injr' } i) &= \texttt{[19]}_{2^8} \cdot \text{byteCode}(k-1, i-2) && \text{where } 1 < i \\
\text{byteCode}(k, \text{`take' } i) &= \texttt{[1a]}_{2^8} \cdot \text{byteCode}(k-1, i-2) && \text{where } 1 < i \\
\text{byteCode}(k, \text{`drop' } i) &= \texttt{[1b]}_{2^8} \cdot \text{byteCode}(k-1, i-2) && \text{where } 1 < i \\
\text{byteCode}(k, \text{`iden'}) &= \texttt{[20]}_{2^8} \\
\text{byteCode}(k, \text{`unit'}) &= \texttt{[21]}_{2^8} \\
\text{byteCode}(k, \text{`fail' } b) &= \texttt{[22]}_{2^8} \cdot \text{BE}(b) \\
\text{byteCode}(k, \text{`hidden' } h) &= \texttt{[23]}_{2^8} \cdot \text{BE}(h) \\
\text{byteCode}(k, \text{`witness' } v) &= \text{BE}(\lfloor \texttt{80}_{2^8} + |v| \rfloor_8) \cdot \text{bytePad}(v) && \text{where } |v| < 127 \\
\text{byteCode}(k, \text{`witness' } v) &= \texttt{[ff]}_{2^8} \cdot \text{BE}(\lfloor |v| \rfloor_{16}) \cdot \text{bytePad}(v) && \text{where } 127 \leq |v| < 2^{16}
\end{aligned}$$

The byte codes for primitives begin with a byte between $\texttt{[24]}_{2^8}$ and $\texttt{[3f]}_{2^8}$ inclusive. The codes can contain multiple bytes. However, for the Bitcoin primitives, we only need to use one byte per primitive.

$$\begin{aligned}
\text{byteCode}(k, \text{`version'}) &= \texttt{[24]}_{2^8} \\
\text{byteCode}(k, \text{`lockTime'}) &= \texttt{[25]}_{2^8} \\
\text{byteCode}(k, \text{`inputsHash'}) &= \texttt{[26]}_{2^8} \\
\text{byteCode}(k, \text{`outputsHash'}) &= \texttt{[27]}_{2^8} \\
\text{byteCode}(k, \text{`numInputs'}) &= \texttt{[28]}_{2^8} \\
\text{byteCode}(k, \text{`totalInputValue'}) &= \texttt{[29]}_{2^8} \\
\text{byteCode}(k, \text{`currentPrevOutpoint'}) &= \texttt{[2a]}_{2^8} \\
\text{byteCode}(k, \text{`currentValue'}) &= \texttt{[2b]}_{2^8} \\
\text{byteCode}(k, \text{`currentSequence'}) &= \texttt{[2c]}_{2^8} \\
\text{byteCode}(k, \text{`currentIndex'}) &= \texttt{[2d]}_{2^8} \\
\text{byteCode}(k, \text{`inputPrevOutpoint'}) &= \texttt{[2e]}_{2^8} \\
\text{byteCode}(k, \text{`inputValue'}) &= \texttt{[2f]}_{2^8} \\
\text{byteCode}(k, \text{`inputSequence'}) &= \texttt{[30]}_{2^8} \\
\text{byteCode}(k, \text{`numOutputs'}) &= \texttt{[31]}_{2^8} \\
\text{byteCode}(k, \text{`totalOutputValue'}) &= \texttt{[32]}_{2^8} \\
\text{byteCode}(k, \text{`outputValue'}) &= \texttt{[33]}_{2^8} \\
\text{byteCode}(k, \text{`outputScriptHash'}) &= \texttt{[34]}_{2^8} \\
\text{byteCode}(k, \text{`scriptCMR'}) &= \texttt{[35]}_{2^8}
\end{aligned}$$

We define a byte-string prefix code for Simplicity DAGs as a concatenation of byte-string codes of its nodes, terminated by a sentinel value that has been reserved as an end-of-stream byte.

$$\text{byteCode}(l) := ((\mu^* \circ \eta^{\text{S}} \circ \text{byteCode}^+ \circ \text{indexed})(l)) \cdot \texttt{[1f]}_{2^8}$$

Notice that while $\text{byteCode}(0, x)$ for $x : \text{Node}$ seems like it could call $\text{byteCode}(-1, n)$ for some $n : \mathbb{N}$, this can never happen for Simplicity DAGs. Recall from Section 7.1, that a well-formed Simplicity DAG, $l : \text{Node}^+$, satisfies the condition

$$\forall \langle i, a \rangle \in \text{indexed}(l).\, \forall j \in \text{ref}(a).\, 0 < j \leq i.$$

The condition on DAGs implies that $|\text{ref}(l[0])| = 0$. This condition means that $\text{byteCode}(-1, n)$ never occurs for any $n : \mathbb{N}$.

Glossary

Bibliography