Simplicity

Russell O'Connor

Blockstream

Email: `roconnor@blockstream.com`

November 19, 2018

DRAFT

Table of contents

1 Introduction . . . 7

1.1 Bitcoin Script . . . 7
1.2 Simplicity's Design Goals . . . 8
1.2.1 Static Analysis . . . 8
1.2.2 Pruning and Sharing . . . 8
1.2.3 Formal Semantics . . . 9

2 Type Theory Preliminaries . . . 11

2.1 Algebraic Types . . . 11
2.1.1 Records . . . 12
2.2 Functors . . . 13
2.2.1 Option Functor . . . 13
2.2.2 List Functors . . . 13
2.3 Monads . . . 15
2.3.1 Kleisli Morphisms . . . 15
2.3.2 Cartesian Strength . . . 15
2.3.3 Identity Monad . . . 16
2.3.4 Monad Zero . . . 16
2.3.4.1 Option Monad . . . 17
2.4 Multi-bit Words . . . 17
2.4.1 Byte Strings . . . 18
2.4.2 Bit Strings . . . 18

3 Core Simplicity . . . 19

3.1 Types . . . 19
3.1.1 Abstract Syntax . . . 19
3.1.2 Formal Syntax . . . 19
3.1.3 Formal Semantics . . . 19
3.2 Terms . . . 20
3.2.1 Identity . . . 20
3.2.2 Composition . . . 20
3.2.3 Constant Unit . . . 20
3.2.4 Left Injection . . . 20
3.2.5 Right Injection . . . 20
3.2.6 Case . . . 21
3.2.7 Pair . . . 21
3.2.8 Take . . . 21
3.2.9 Drop . . . 21
3.2.10 Formal Syntax . . . 21
3.2.11 Formal Semantics . . . 22
3.3 Example Simplicity . . . 22
3.3.1 Bit Operations . . . 22
3.3.2 Simplicity Notation . . . 23
3.3.3 Arithmetic . . . 23
3.3.4 Bitwise Operations . . . 26

3.3.5 SHA-256 . . . 26
3.3.6 Elliptic Curve Operations on secp256k1 . . . 27
3.3.6.1 libsecp256k1 . . . 27
3.3.6.2 libsecp256k1 in Simplicity . . . 28
3.3.6.3 Schnorr Signature Validation . . . 30
3.4 Completeness Theorem . . . 30
3.5 Operational Semantics . . . 31
3.5.1 Repesenting Values as Cell Arrays . . . 31
3.5.2 Bit Machine . . . 31
3.5.2.1 Frame Instructions . . . 32
3.5.2.2 Active Write Frame Instructions . . . 32
3.5.2.3 Active Read Frame Instructions . . . 33
3.5.2.4 Abort Instruction . . . 33
3.5.2.5 Bit Machine Programs . . . 33
3.5.2.6 Crashing the Bit Machine . . . 34
3.5.3 Executing Simplicity . . . 34
3.5.3.1 Tail Composition Optimisation (TCO) . . . 35
3.6 Static Analysis . . . 36
3.6.1 Space Resources . . . 36
3.6.1.1 Maximum Cell Count Bound . . . 36
3.6.1.2 Maximum Frame Count Bound . . . 39
3.6.2 Time Resources . . . 39
3.7 Commitment Merkle Root . . . 39

4 Simplicity Extensions . . . 41

4.1 Monadic Semantics . . . 41
4.2 Witness . . . 42
4.2.1 Elided Computation . . . 42
4.2.2 Witness Merkle Root . . . 42
4.2.3 Type Inference with Witness . . . 42
4.3 Assertions and Failure . . . 42
4.3.1 Denotational Semantics . . . 43
4.3.2 Merkle Roots . . . 43
4.3.2.1 Pruning Unused case Branches . . . 44
4.3.2.2 Salted Expressions . . . 44
4.4 Blockchain Primitives . . . 45
4.4.1 Bitcoin Transactions . . . 45
4.4.1.1 Denotational Semantics . . . 47
4.4.1.2 Merkle Roots . . . 48
4.5 Simplicity Programs . . . 48
4.5.1 Example: checkSigHashAll . . . 48
4.6 Schnorr Signature Aggregation . . . 49
4.7 Malleability . . . 49
4.7.1 Transaction Weight . . . 49

5 Jets . . . 51

6 Delegation . . . 53

6.1 Unbounded Loops . . . 53

7 Type Inference and Serialization . . . 55

7.1 Explicit Simplicity DAGs . . . 55
7.1.1 Type Inference . . . 56
7.1.2 Reconstructing a Simplicity Expressions . . . 58

7.1.2.1 syncase 59
7.1.2.2 inflate 59
7.2 Serialization 60
7.2.1 Serializtion of Bit String and Positive Numbers 60
7.2.2 Serialization of Simplicity 61

8 Coq Library Guide 63

8.1 Simplicity Types 63
8.2 Simplicity Terms 63
8.2.1 The “Initial” Representation of Terms 63
8.2.2 The “Final” Representation of Terms 63
8.2.2.1 Simplicity Algebras 64
8.2.2.2 The “Final” Representation 64
8.2.2.3 Constructing “Final” Terms 65
8.2.3 Why two representations of Terms? 65
8.3 Example Simplicity Expressions 65
8.3.1 Bits 65
8.3.2 Arithmetic 65
8.3.3 SHA256 66
8.4 The Hierarchy of Simplicity Language Extensions 66
8.4.1 Witness 67
8.4.2 Assertion 67
8.4.3 Delegation 67
8.4.4 Primitives 68
8.4.4.1 Bitcoin 68
8.4.5 Jets 68
8.4.6 Full Simplicity 69
8.5 Merkle Roots 69
8.6 The Bit Machine 69
8.6.1 Bit Machine Code 70
8.6.1.1 Bit Machine Programs 70
8.6.2 Translating Simplicity to the Bit Machine 71
8.6.3 Static Analysis 71

9 Haskell Library Guide 73

9.1 Simplicity Types 73
9.2 Simplicity Terms 74
9.3 Blockchain Primitives 75
9.3.1 Bitcoin Primitives 75
9.4 Merkle Roots 75
9.5 Denotational Semantics of Full Simplicity 75
9.6 Example Simplicity Expressions 76
9.6.1 Bits 76
9.6.2 Multi-bit Words 76
9.6.2.1 Arithmetic operations 76
9.6.2.2 Bit-wise operations 76
9.6.3 Generic 76
9.6.4 SHA-256 76
9.6.5 LibSecp256k1 76
9.6.6 CheckSigHashAll 77
9.7 The Bit Machine 77
9.7.1 Translating Simplicity to the Bit Machine 77
9.7.2 Static Analysis 78
9.8 Type Inference 78

9.9 Serialization 78
9.9.1 Free Monadic Deserializaiton 78
9.9.2 Serialization of Simplicity DAGs 80

10 C Library Guide 81

Appendix A Alternative Serialization of Simplicity DAGs 83

Bibliography 85

Chapter 1
Introduction

1.1 Bitcoin Script

Bitcoin[11] was the first protocol that used a blockchain to build a distributed ledger that allows anyone to transact a cyrptographic 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 cyrptographic 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*[12, 2]. 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 cyrptographic 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 [5] 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 being 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`[cafe]` instead of f(`[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

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

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:

$$\lambda x\colon A.\,e:A\to B$$

where e is an expression with $x : A$ as a free 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 $\mathrm{id}_A : A \to A$ by

$$\mathrm{id}_A(a) := a$$

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

$$(g \circ f)(a) := g(f(a))$$

To access components of sum and product types we use pattern matching. For example, we define the first and second projection functions as

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

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:

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

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 $\mathtt{1}_{\mathbb{2}} : \mathbb{2}$.

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

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.

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

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

$$\begin{gathered} r[\mathrm{field}_1] : A_1 \\ r[\mathrm{field}_2] : A_2 \\ \vdots \\ r[\mathrm{field}_n] : A_n \end{gathered}$$

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

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

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

$$\begin{aligned} \mathcal{F}\mathrm{id}_A &= \mathrm{id}_{\mathcal{F}A} \\ \mathcal{F}(f \circ g) &= \mathcal{F}f \circ \mathcal{F}g \end{aligned}$$

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,

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

where $f : A \to B$.

We define special notation for values of "optional" types $\mathrm{S}A$,

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

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.

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

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.

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

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

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

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 $l : A^+$.

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

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*, $a \blacktriangleleft l : A^*$

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

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

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

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

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

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

$$\mu^*_A(l) := \mathrm{fold}^{\langle \cdot, \epsilon \rangle}_{A^*}(l)$$

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

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

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

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

2.3 Monads

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

$$\begin{aligned} \eta^{\mathcal{M}}_A &: A \to \mathcal{M}\, A \\ \mu^{\mathcal{M}}_A &: \mathcal{M}\, \mathcal{M}\, A \to \mathcal{M}\, A \end{aligned}$$

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

$$\begin{aligned} \mathcal{M}\, f \circ \eta^{\mathcal{M}}_A &= \eta^{\mathcal{M}}_B \circ f \\ \mathcal{M}\, f \circ \mu^{\mathcal{M}}_A &= \mu^{\mathcal{M}}_B \circ \mathcal{M}\, \mathcal{M}\, f \end{aligned}$$

The $\eta^{\mathcal{M}}_A$ and $\mu^{\mathcal{M}}_A$ 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 $g \overset{\mathcal{M}}{\hookleftarrow} f : A \to \mathcal{M}\, C$ where

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

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:

$$\begin{aligned} f \hookleftarrow \eta^{\mathcal{M}}_A &= f \\ \eta^{\mathcal{M}}_B \hookleftarrow f &= f \\ (h \hookleftarrow g) \hookleftarrow f &= h \hookleftarrow (g \hookleftarrow f) \end{aligned}$$

2.3.2 Cartesian Strength

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

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

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

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

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

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

It is always the case that

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

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 $\mathcal{M}\, \Delta_A = \phi^{\mathcal{M}}_{A,A} \circ \Delta_{\mathcal{M} A}$ (equiv. $\mathcal{M}\, \Delta_A = \bar{\phi}^{\mathcal{M}}_{A,A} \circ \Delta_{\mathcal{M} A}$), 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^{Id}_A and μ^{Id}_A are both the identity function. The identity monad captures no side-effects and it is commutative and idempotent.

2.3.4 Monad Zero

Some monads have a universal *zero* value

$$\emptyset^{\mathcal{M}}_A : \mathcal{M}\, A$$

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

$$\mathcal{M} f(\emptyset^{\mathcal{M}}_A) = \emptyset^{\mathcal{M}}_B$$

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 $\varnothing^{\mathcal{M}}_{A,B} : A \to \mathcal{M}\, B$ as a *zero morphism*.

$$\varnothing^{\mathcal{M}}_{A,B}(a) := \emptyset^{\mathcal{M}}_B$$

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

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

2.1. Beware that we are using the definition of idempotent monad from King and Wadler [8], as opposed to the traditional categorical definition of an idempotent monad.

(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:

$$\begin{array}{rcl} \eta^{\mathrm{S}}_A(a) & := & \sigma^{\mathbf{R}}(a) \\ \mu^{S}_A(\sigma^{\mathbf{L}}\langle\rangle) & := & \sigma^{\mathbf{L}}\langle\rangle \\ \mu^{S}_A(\sigma^{\mathbf{R}}(x)) & := & x \end{array}$$

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

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

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

$$\begin{array}{rcl} \iota^{\mathcal{M}}_{\mathrm{S},A}(\emptyset^{\mathrm{S}}_A) & := & \emptyset^{\mathcal{M}}_A \\ \iota^{\mathcal{M}}_{\mathrm{S},A}(\eta^{\mathrm{S}}_A(a)) & := & \eta^{\mathcal{M}}_A(a) \end{array}$$

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

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

Also

$$\iota^{\mathcal{M}}_{\mathrm{S},A} \circ \mu^{\mathrm{S}}_A = \mu^{\mathcal{M}}_A \circ \iota^{\mathcal{M}}_{\mathrm{S},\mathcal{M}A} \circ \mathrm{S}\iota^{\mathcal{M}}_{\mathrm{S},A}\ (= \mu^{\mathcal{M}}_A \circ \mathcal{M}\,\iota^{\mathcal{M}}_{\mathrm{S},A} \circ \iota^{\mathcal{M}}_{\mathrm{S},\mathrm{S}A})$$

2.4 Multi-bit Words

By repeatedly taking products of the bit type we can build the types $\mathbb{2}^{2^n}$ 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 $a : \mathbb{2}^n$, where n is a power of two, we recursively define $\lceil a \rceil_n : \mathbb{N}$ to be the number that a represents:

$$\begin{array}{rcl} \lceil \mathtt{0}_{\mathbb{2}} \rceil_1 & := & 0 \\ \lceil \mathtt{1}_{\mathbb{2}} \rceil_1 & := & 1 \\ \lceil \langle a, b\rangle \rceil_{2n} & := & \lceil a \rceil_n\, 2^n + \lceil b \rceil_n \end{array}$$

We also make use of the following variation of this value intepretation 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 $\lfloor m \rfloor_n : \mathbb{2}^n$ 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 $\mathbb{2}^n$ with addition and multiplication operations, so that $\lfloor \cdot \rfloor_n$ becomes a semiring homomorphism. Given $a : \mathbb{2}^n$ and $b : \mathbb{2}^n$, we define $a \lfloor + \rfloor_n b : \mathbb{2}^n$ and $a\ \lfloor \times \rfloor_n b : \mathbb{2}^n$ such that

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

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

2.4.1 Byte Strings

The type $(\mathbb{2}^8)^*$ is known as byte strings. We will write byte string values as sequences of hexadecimal digits surrounded by square brackets, e.g. $\texttt{[cafe]}_{\mathbb{2}^8}$ denotes $\texttt{ca}_{\mathbb{2}^8} \blacktriangleleft \texttt{fe}_{\mathbb{2}^8} \blacktriangleleft \epsilon_{\mathbb{2}^8}$ (whereas $\texttt{cafe}_{\mathbb{2}^{16}}$ denotes $\langle \texttt{ca}_{\mathbb{2}^8}, \texttt{fe}_{\mathbb{2}^8}\rangle$). 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 $\mathrm{BE}_n : \mathbb{2}^n \to (\mathbb{2}^8)^*$ and $\mathrm{LE}_n : \mathbb{2}^n \to (\mathbb{2}^8)^*$ 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 $\mathbb{2}^*$ is known as bit strings. We will write bit string values as sequences of binary digits surrounded by square brackets, e.g. $\texttt{[0110]}_{\mathbb{2}}$ denotes $\texttt{0}_{\mathbb{2}} \blacktriangleleft \texttt{1}_{\mathbb{2}} \blacktriangleleft \texttt{1}_{\mathbb{2}} \blacktriangleleft \texttt{0}_{\mathbb{2}} \blacktriangleleft \epsilon_{\mathbb{2}}$. 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 [7]. 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 $\mathbb{2}$, 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 $[\![t]\!] : A \rightarrow B$ for the function that the term t denotes.

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

3.2.1 Identity

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

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

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

3.2.2 Composition

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

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

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

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

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

The constant unit term ignores its argument and always returns $\langle\rangle$, the unique value of the unit type. The argument is ignored so we have a constant unit term for every type.

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, $\mathbb{2}$, we can define core Simplicity terms that represent the two constant functions that return this type:

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

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

$$\begin{aligned}[\![\mathsf{false}]\!](a) &= \mathsf{0}_{\mathbb{2}}\\ [\![\mathsf{true}]\!](a) &= \mathsf{1}_{\mathbb{2}}\end{aligned}$$

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.

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

We can prove that cond has the following semantics.

$$\begin{array}{rcl} [\![\mathsf{cond}\, s t]\!]\langle 1_{2}, a\rangle &=& [\![s]\!](a) \\ [\![\mathsf{cond}\, s t]\!]\langle 0_{2}, a\rangle &=& [\![t]\!](a) \end{array}$$

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

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

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

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

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: $\mathbb{2} \times \mathbb{2} \vdash \mathbb{2}$ and or-func: $\mathbb{2} \times \mathbb{2} \vdash \mathbb{2}$, 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.

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

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.

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

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

3.3.3 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 adder and full adder.

$$\begin{array}{l} \mathsf{adder}_1 : \mathbb{2} \times \mathbb{2} \vdash \mathbb{2}^2 \\ \quad \mathsf{adder}_1 \;:=\; \mathsf{cond}\, (\mathsf{iden} \vartriangle \mathsf{not}\, \mathsf{iden})\, (\mathsf{false} \vartriangle \mathsf{iden}) \end{array}$$

$$\begin{array}{l} \mathsf{full\text{-}adder}_1 : (\mathbb{2} \times \mathbb{2}) \times \mathbb{2} \vdash \mathbb{2}^2 \\ \quad \begin{array}{rcl} \mathsf{full\text{-}adder}_1 &:=& \mathsf{take}\, \mathsf{adder}_1 \vartriangle \mathsf{IH} \\ &;& \mathsf{OOH} \vartriangle (\mathsf{OIH} \vartriangle \mathsf{IH}; \mathsf{adder}_1) \\ &;& \mathsf{cond}\, \mathsf{true}\, \mathsf{OH} \vartriangle \mathsf{IIH} \end{array} \end{array}$$

These adders meet the following specifications.

$$\begin{aligned}\lceil[\![\mathsf{adder}_1]\!]\langle a,b\rangle\rceil_2 &= \lceil a\rceil_1+\lceil b\rceil_1\\ \lceil[\![\mathsf{full\text{-}adder}_1]\!]\langle\langle a,b\rangle,c\rangle\rceil_2 &= \lceil a\rceil_1+\lceil b\rceil_1+\lceil c\rceil_1\end{aligned}$$

It is easy to exhaustively check the above equations because there are only a small finite number of possible inputs to consider (four inputs for adder_1 and eight inputs for $\mathsf{full\text{-}adder}_1$). We will illustrate this for a single case for adder_1 where $a = \mathsf{1}_2$ and $b = \mathsf{0}_2$.

$$\begin{aligned}\lceil[\![\mathsf{adder}_1]\!]\langle \mathsf{1}_2,\mathsf{0}_2\rangle\rceil_2 &= \lceil[\![\mathsf{cond}\ (\mathsf{iden} \vartriangle \mathsf{not\ iden})\ (\mathsf{false} \vartriangle \mathsf{iden})]\!]\langle \mathsf{1}_2,\mathsf{0}_2\rangle\rceil_2\\ &= \lceil[\![\mathsf{iden} \vartriangle \mathsf{not\ iden}]\!](\mathsf{0}_2)\rceil_2\\ &= \lceil\langle[\![\mathsf{iden}]\!](\mathsf{0}_2), [\![\mathsf{not\ iden}]\!](\mathsf{0}_2)\rangle\rceil_2\\ &= \lceil\langle \mathsf{0}_2, [\![(\mathsf{pair\ iden\ unit}); (\mathsf{cond\ false\ true})]\!](\mathsf{0}_2)\rangle\rceil_2\\ &= \lceil\langle \mathsf{0}_2, [\![(\mathsf{cond\ false\ true})]\!] \circ [\![(\mathsf{pair\ iden\ unit})]\!](\mathsf{0}_2)\rangle\rceil_2\\ &= \lceil\langle \mathsf{0}_2, [\![(\mathsf{cond\ false\ true})]\!]\langle[\![\mathsf{iden}]\!](\mathsf{0}_2), [\![\mathsf{unit}]\!](\mathsf{0}_2)\rangle\rangle\rceil_2\\ &= \lceil\langle \mathsf{0}_2, [\![(\mathsf{cond\ false\ true})]\!]\langle \mathsf{0}_2, \langle\rangle\rangle\rangle\rceil_2\\ &= \lceil\langle \mathsf{0}_2, [\![\mathsf{true}]\!]\langle\rangle\rangle\rceil_2\\ &= \lceil\langle \mathsf{0}_2, \mathsf{1}_2\rangle\rceil_2\\ &= \lceil \mathsf{0}_2\rceil_1\cdot 2^1+\lceil \mathsf{1}_2\rceil_1\\ &= 0\cdot 2+1\\ &= 1\\ &= 1+0\\ &= \lceil \mathsf{1}_2\rceil_1+\lceil \mathsf{0}_2\rceil_1\end{aligned}$$

The calculations for the other cases are similar.

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

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

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

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

$$\begin{aligned}\lceil[\![\mathsf{adder}_n]\!]\langle a,b\rangle\rceil_{1,n} &= \lceil a\rceil_n+\lceil b\rceil_n\\ \lceil[\![\mathsf{full\text{-}adder}_n]\!]\langle\langle a,b\rangle,c\rangle\rceil_{1,n} &= \lceil a\rceil_n+\lceil b\rceil_n+\lceil c\rceil_1\end{aligned}$$

Theorem 3.1. *For all n which is a power of 2, and for all a: $\mathbb{2}^n$, b: $\mathbb{2}^n$, and c: $\mathbb{2}$, we have that $\lceil[\![\mathsf{full\text{-}adder}_n]\!]\langle\langle a, b\rangle, c\rangle\rceil_{1,n} = \lceil a\rceil_n+\lceil b\rceil_n+\lceil c\rceil_1$.*

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

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

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

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

so we know that

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

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

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

therefore we have that

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

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

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

Therefore we have that

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

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. □

The proof that $\lceil [\![\mathsf{adder}_n]\!] \langle a, b \rangle \rceil_{1,n} = \lceil a \rceil_n + \lceil b \rceil_n$ is done in a similar manner. 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 multipliers and full multiplier.

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

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

$$\begin{aligned}
&\mathsf{multiplier}_1 : 2 \times 2 \vdash 2^2\\
&\mathsf{multiplier}_1 \;:=\; \mathsf{false} \vartriangle \mathsf{cond}\;\mathsf{iden}\;\mathsf{false}
\end{aligned}$$

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

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

$$\begin{aligned}
\lceil [\![\mathsf{full\text{-}multiplier}_n]\!] \langle \langle a, b \rangle, \langle c, d \rangle \rangle \rceil_{2n} &= \lceil a \rceil_n \cdot \lceil b \rceil_n + \lceil c \rceil_n + \lceil d \rceil_n\\
\lceil [\![\mathsf{multiplier}_n]\!] \langle a, b \rangle \rceil_{2n} &= \lceil a \rceil_n \cdot \lceil b \rceil_n
\end{aligned}$$

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

3.3.4 Bitwise Operations

3.3.5 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) [13]. 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 $\mathrm{SHA256}_{\mathrm{MD}} : 2^{256} \to (2^{512})^* \to 2^{256}$.

$$\mathrm{SHA256}_2 = \mathrm{SHA256}_{\mathrm{MD}}(\mathrm{SHA256}_{\mathrm{IV}}) \circ \eta^S \circ \mathrm{SHA256}_{\mathrm{Pad}}$$

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 $\mathrm{SHA256}_{\mathrm{Block}} : 2^{256} \times 2^{512} \to 2^{256}$.

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

The block compression function $\mathrm{SHA256_{Block}} : 2^{256} \times 2^{512} \to 2^{256}$ 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

$$[\![\mathsf{sha256\text{-}block}]\!] = \mathrm{SHA256_{Block}}$$

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

$$[\![\mathsf{sha256\text{-}iv}]\!]\langle\rangle = \mathrm{SHA256_{IV}}$$

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 $\mathrm{SHA256}_{2^8} : (2^8)^* \to 2^{256}$.

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

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 Elliptic Curve Operations on secp256k1

The Standards for Efficient Cryptography (SEC) documents have recommend modular elliptic curve parameters including the secp256k1 curve [4], 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.

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

3.3.6.1 libsecp256k1

The libsecp256k1 library [18] is a C implementation of optimized functions on this elliptic curve. This library has two variants for the representation for field elements, of which the `10x26` representation is the most portable. This representation consists of an array of 10 32-bit unsigned integer values. Such an array, `a`, represents the value

$$\sum_{i=0}^{9} \texttt{a[}i\texttt{]} \cdot 2^{26i} \pmod{p}.$$

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.

Different arrays may represent the same value. The various field arithmetic operations, including modular inverse and square roots, are implemented efficiently, subject to various specific preconditions on their inputs that need to be satisfied to prevent overflows of the internal 32-bit unsigned integer values.

The libsecp256k1 library also 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).

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.

3.3.6.2 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

$$\mathrm{FE} := 2^{32} \times (2^{32} \times (2^{32} \times (2^{32} \times (2^{32} \times (2^{32} \times (2^{32} \times (2^{32} \times (2^{32} \times 2^{32}))))))))$$

and a value $\langle a_0, \langle a_1, \langle a_2, \langle a_3, \langle a_4, \langle a_5, \langle a_6, \langle a_7, \langle a_8, a_9\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle : \mathrm{FE}$ represents the field element

$$\lceil\langle a_0, \langle a_1, \langle a_2, \langle a_3, \langle a_4, \langle a_5, \langle a_6, \langle a_7, \langle a_8, a_9\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rceil_{\mathrm{FE}} := \sum_{i=0}^{9} \lceil a_i\rceil_{32} \cdot 2^{26i} \pmod{p}.$$

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

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

The translation between the libsecp256k1's `10x32` field representation and Simplicity's FE type is straightforward. The translation between libsecp256k1's affine coordinate representation of elliptic curve points and Simplicity's GE is also 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 field elements, or points, the Simplicity expression returns the exact same representative for its result as libsecp256k1. If a precondition of a libsecp256k1 function is violated, the the Simplicity code also overflows and returns the corresponding value that libsecp256k1 returns. 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.

3.3.6.3 Schnorr Signature Validation

With elliptic curve operations defined, we are able to implement Schnorr signature validation in accordance with the BIP-Schnorr specification [17]. We define Simplicity types for the formats of compressed public keys, PubKey; messages, Msg; and Schnorr signatures, Sig, below.

$$\begin{aligned}\text{PubKey} &:= \mathbb{2} \times \mathbb{2}^{256} \\ \text{Msg} &:= \mathbb{2}^{256} \\ \text{Sig} &:= \mathbb{2}^{512}\end{aligned}$$

The PubKey type is a pair $\langle b, \lfloor x \rfloor_{256} \rangle$ where b is the least significant bit of a (non-$\mathcal{O}$) elliptic curve point's y-coordinate, and where x the point's x-coordinate. A Msg value m represents the byte-string $\text{BE}_{256}(m)$ for a Schnorr signature's message, and a Sig value a represents the byte-string $\text{BE}_{512}(a)$ for a Schnorr signature.

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

$$\mathsf{schnorrVerify} : (\text{PubKey} \times \text{Msg}) \times \text{Sig} \vdash \mathbb{2}$$

The semantics are such that $[\![\mathsf{schnorrVerify}]\!]\langle\langle p, m\rangle, s\rangle = 1_{\mathbb{2}}$ only when the values that the inputs represents satisfy the verification conditions of the BIP-Schnorr 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 3.2. *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 3.3. *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 Repesenting 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.

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

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

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

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

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

Theorem 3.4. *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.

$$\texttt{[0}\underline{\texttt{1}}\texttt{?10]}$$

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, $\texttt{[}\underline{\texttt{]}}$, 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
$\texttt{[100}\underline{\texttt{1}}\texttt{1??110101000]}$	$\texttt{[11??1101}\underline{\texttt{]}}$
$\texttt{[}\underline{\texttt{0}}\texttt{000]}$	$\texttt{[111}\underline{\texttt{?}}\texttt{?]}$
$\texttt{[}\underline{\texttt{]}}$	
$\texttt{[}\underline{\texttt{1}}\texttt{0]}$	

Figure 3.1. Example state of the Bit Machine.

Notationally we will write a stack of read frames as $r_n \triangleright \ldots \triangleright r_1 \triangleright r_0$, with r_0 as the active read frame. We will write a stack of write frames in the opposite order, as $w_0 \triangleleft w_1 \triangleleft \ldots \triangleleft w_m$ with w_0 as the active write frame. We write a state of the Bit Machine as $[\Theta \triangleright r_0 | w_0 \triangleleft \Xi]$ 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.

$$\begin{aligned}
\text{newFrame}(n) &: [\Theta \triangleright r_0 | w_0 \triangleleft \Xi] \rightsquigarrow [\Theta \triangleright r_0 | {}_\wedge \texttt{[?]}^n \triangleleft w_0 \triangleleft \Xi] \\
\text{moveFrame} &: [\Theta \triangleright r_0 | \texttt{[}c_1 \cdots c_n\underline{\texttt{]}} \triangleleft w_0 \triangleleft \Xi] \rightsquigarrow [\Theta \triangleright r_0 \triangleright \texttt{[}\underline{c_1} \cdots c_n\texttt{]} | w_0 \triangleleft \Xi] \\
\text{dropFrame} &: [\Theta \triangleright r_1 \triangleright r_0 | \Xi] \rightsquigarrow [\Theta \triangleright r_1 | \Xi]
\end{aligned}$$

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.

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

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.

[3.2]. 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).

3.5.2.3 Active Read Frame Instructions

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

$$\text{fwd}(n): [\Theta \rhd r_0 \cdot \texttt{[}\underline{\texttt{c}_1}\cdots\ \texttt{c}_n\texttt{]} \cdot r'_0 | w_0 \lhd \Xi] \rightsquigarrow [\Theta \rhd r_0 \cdot \texttt{[}\texttt{c}_1 \cdots\ \texttt{c}_n\underline{\texttt{]}} \cdot r'_0 | w_0 \lhd \Xi]$$

$$\text{bwd}(n): [\Theta \rhd r_0 \cdot \texttt{[}\texttt{c}_1 \cdots\ \texttt{c}_n\underline{\texttt{]}} \cdot r'_0 | w_0 \lhd \Xi] \rightsquigarrow [\Theta \rhd r_0 \cdot \texttt{[}\underline{\texttt{c}_1}\cdots\ \texttt{c}_n\texttt{]} \cdot r'_0 | w_0 \lhd \Xi]$$

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 $S_0 \rightarrowtail k \twoheadrightarrow S_1$ for a program, k, that, when executed, successfully transforms an initial state S_0 to the final state S_1.

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

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

$$\frac{i \colon S_0 \rightsquigarrow S_1}{S_0 \rightarrowtail i \twoheadrightarrow S_1}$$

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

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

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.

$$\frac{[\Theta \rhd r_0 \cdot \texttt{[}\underline{\texttt{0}}\texttt{]} \cdot r'_0 | w_0 \lhd \Xi] \rightarrowtail k_0 \twoheadrightarrow S}{[\Theta \rhd r_0 \cdot \texttt{[}\underline{\texttt{0}}\texttt{]} \cdot r'_0 | w_0 \lhd \Xi] \rightarrowtail k_0 || k_1 \twoheadrightarrow S}$$

$$\frac{[\Theta \rhd r_0 \cdot \texttt{[}\underline{\texttt{1}}\texttt{]} \cdot r'_0 | w_0 \lhd \Xi] \rightarrowtail k_1 \twoheadrightarrow S}{[\Theta \rhd r_0 \cdot \texttt{[}\underline{\texttt{1}}\texttt{]} \cdot r'_0 | w_0 \lhd \Xi] \rightarrowtail k_0 || k_1 \twoheadrightarrow S}$$

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.

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

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 3.5.

$$\frac{[\Theta \triangleright r_0 \cdot \mathtt{[}c_1 \cdots c_n\underline{\;}\mathtt{]} \cdot r'_0 | w_0 \triangleleft \Xi] \rightarrowtail k \twoheadrightarrow [\Theta \triangleright r_0 \cdot \mathtt{[}c_1 \cdots c_n\underline{\;}\mathtt{]} \cdot r'_0 | w'_0 \triangleleft \Xi']}{[\Theta \triangleright r_0 \cdot \mathtt{[}\underline{c_1} \cdots c_n\mathtt{]} \cdot r'_0 | w_0 \triangleleft \Xi] \rightarrowtail n \star k \twoheadrightarrow [\Theta \triangleright r_0 \cdot \mathtt{[}\underline{c_1} \cdots c_n\mathtt{]} \cdot r'_0 | w'_0 \triangleleft \Xi']}$$

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:

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

Theorem 3.6. *Given a well-typed core Simplicity program $t : A \vdash B$ and an input $a : A$, then*

$$\left[\Theta \triangleright r_0 {}_\wedge \ulcorner a \urcorner \cdot r'_0 | w_0 {}_\wedge \mathtt{[?]}^{\text{bitSize}(B)+m} \triangleleft \Xi\right] \rightarrowtail \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [\Theta \triangleright r_0 {}_\wedge \ulcorner a \urcorner \cdot r'_0 | w_0 \cdot \ulcorner [\![t]\!](a) \urcorner {}_\wedge \mathtt{[?]}^m \triangleleft \Xi]$$

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

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

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 $\langle\!\langle\cdot\rangle\!\rangle^{\mathrm{TCO}}_{\mathrm{on}}$:

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

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

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

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

and

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

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 $t: A \vdash B$, we have

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

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.

$$\begin{aligned}
\mathrm{cellCount}(\texttt{[}c_1 \cdots \underline{c_i} \cdots c_n\texttt{]}) &:= n \\
\mathrm{cellCount}([r_n \triangleright \ldots \triangleright r_0 | w_0 \triangleleft \ldots \triangleleft w_m]) &:= \sum_{i=0}^{n} \mathrm{cellCount}(r_i) + \sum_{j=0}^{m} \mathrm{cellCount}(w_j) \\
\mathrm{cellCount}(\boxtimes) &:= 0
\end{aligned}$$

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

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

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

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

$$\frac{[\Theta \triangleright r_0 \cdot \texttt{[}\underline{\texttt{0}}\texttt{]} \cdot r_0' | w_0 \triangleleft \Xi] \rightarrowtail k_0 \twoheadrightarrow S}{\mathrm{cellsReq}([\Theta \triangleright r_0 \cdot \texttt{[}\underline{\texttt{0}}\texttt{]} \cdot r_0' | w_0 \triangleleft \Xi] \rightarrowtail k_0 || k_1 \twoheadrightarrow S) := \mathrm{cellsReq}([\Theta \triangleright r_0 \cdot \texttt{[}\underline{\texttt{0}}\texttt{]} \cdot r_0' | w_0 \triangleleft \Xi] \rightarrowtail k_0 \twoheadrightarrow S)}$$

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

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

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.

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

Lemma 3.8. *For any core Simplicity expression $t\colon A \vdash B$, such that*

$$[\Theta \triangleright r_0 | w_0 \triangleleft \Xi] \rightarrowtail \langle\!\langle t \rangle\!\rangle \twoheadrightarrow [\Theta' \triangleright r_0' | w_0' \triangleleft \Xi']$$

we have that

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

In particular for $a\colon A$ and

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

we have that

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

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.

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

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

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

we have that

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

The problem with $\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}(t)$ 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 $\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}$ by a pair of parameters.

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

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

$$\begin{array}{rcl}
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(\mathsf{iden}_A) & := & \langle 0, 0\rangle \\
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{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 := \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(s) \\
 & \text{and} & \langle n_t, m_t\rangle := \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t) \\
 & \text{and} & r_b := \mathrm{bitSize}(B) \\
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(\mathsf{unit}_A) & := & \langle 0, 0\rangle \\
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(\mathsf{injl}_{A,B,C}\, t) & := & \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t) \\
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(\mathsf{injr}_{A,B,C}\, t) & := & \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t) \\
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{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 := \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(s) \\
 & \text{and} & \langle n_t, m_t\rangle := \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t) \\
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{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 := \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(s) \\
 & \text{and} & \langle n_t, m_t\rangle := \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t) \\
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(\mathsf{take}_{A,B,C}\, t) & := & \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t) \\
\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(\mathsf{drop}_{A,B,C}\, t) & := & \mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t)
\end{array}$$

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

Lemma 3.10. $\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{dyn}}(t) = \mathrm{interp}^{\mathrm{TCO}}(\mathrm{extraCellsBound}^{\mathrm{TCO}}_{\mathrm{static}}(t))$.

Corollary 3.11. *For any core Simplicity expression $t : A \vdash B$ and $a : A$ such that*

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

we have that

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

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 2.8) we follow the tree structure of a Simplicity expression and compute a commitment Merkle root of its syntax tree. Below we define the commitment Merkle root of a Simplicity expression $t : A \vdash B$ as $\#^c(t) : 2^{256}$.

$$\begin{aligned}
\#^c(\mathsf{iden}_A) &:= \mathrm{tag}^c_{\mathsf{iden}}\\
\#^c(\mathsf{comp}_{A,B,C}\, s\, t) &:= \mathrm{SHA256_{Block}}\langle \mathrm{tag}^c_{\mathsf{comp}}, \langle \#^c(s), \#^c(t)\rangle\rangle\\
\#^c(\mathsf{unit}_A) &:= \mathrm{tag}^c_{\mathsf{unit}}\\
\#^c(\mathsf{injl}_{A,B,C}\, t) &:= \mathrm{SHA256_{Block}}\langle \mathrm{tag}^c_{\mathsf{injl}}, \langle \lfloor 0\rfloor_{256}, \#^c(t)\rangle\rangle\\
\#^c(\mathsf{injr}_{A,B,C}\, t) &:= \mathrm{SHA256_{Block}}\langle \mathrm{tag}^c_{\mathsf{injr}}, \langle \lfloor 0\rfloor_{256}, \#^c(t)\rangle\rangle\\
\#^c(\mathsf{case}_{A,B,C,D}\, s\, t) &:= \mathrm{SHA256_{Block}}\langle \mathrm{tag}^c_{\mathsf{case}}, \langle \#^c(s), \#^c(t)\rangle\rangle\\
\#^c(\mathsf{pair}_{A,B,C}\, s\, t) &:= \mathrm{SHA256_{Block}}\langle \mathrm{tag}^c_{\mathsf{pair}}, \langle \#^c(s), \#^c(t)\rangle\rangle\\
\#^c(\mathsf{take}_{A,B,C}\, t) &:= \mathrm{SHA256_{Block}}\langle \mathrm{tag}^c_{\mathsf{take}}, \langle \lfloor 0\rfloor_{256}, \#^c(t)\rangle\rangle\\
\#^c(\mathsf{drop}_{A,B,C}\, t) &:= \mathrm{SHA256_{Block}}\langle \mathrm{tag}^c_{\mathsf{drop}}, \langle \lfloor 0\rfloor_{256}, \#^c(t)\rangle\rangle
\end{aligned}$$

Here we are directly using SHA-256's compression function, SHA256 Block$\langle i, b\rangle$, which takes two arguments. The first argument, i, is a 256-bit 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 initial values tag^c_x for every combinator by taking the SHA-256 hash of unique byte strings:

$$\begin{aligned}
\mathrm{tag}^c_{\mathsf{iden}} &:= \mathrm{SHA256}\,\texttt{[53696d706c69636974791f436f6d6d69746d656e741f6964656e]}\\
\mathrm{tag}^c_{\mathsf{comp}} &:= \mathrm{SHA256}\,\texttt{[53696d706c69636974791f436f6d6d69746d656e741f636f6d70]}\\
\mathrm{tag}^c_{\mathsf{unit}} &:= \mathrm{SHA256}\,\texttt{[53696d706c69636974791f436f6d6d69746d656e741f756e6974]}\\
\mathrm{tag}^c_{\mathsf{injl}} &:= \mathrm{SHA256}\,\texttt{[53696d706c69636974791f436f6d6d69746d656e741f696e6a6c]}\\
\mathrm{tag}^c_{\mathsf{injr}} &:= \mathrm{SHA256}\,\texttt{[53696d706c69636974791f436f6d6d69746d656e741f696e6a72]}\\
\mathrm{tag}^c_{\mathsf{case}} &:= \mathrm{SHA256}\,\texttt{[53696d706c69636974791f436f6d6d69746d656e741f63617365]}\\
\mathrm{tag}^c_{\mathsf{pair}} &:= \mathrm{SHA256}\,\texttt{[53696d706c69636974791f436f6d6d69746d656e741f70616972]}\\
\mathrm{tag}^c_{\mathsf{take}} &:= \mathrm{SHA256}\,\texttt{[53696d706c69636974791f436f6d6d69746d656e741f74616b65]}\\
\mathrm{tag}^c_{\mathsf{drop}} &:= \mathrm{SHA256}\,\texttt{[53696d706c69636974791f436f6d6d69746d656e741f64726f70]}
\end{aligned}$$

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).

Chapter 4

Simplicity Extensions

The core Simplicity completeness theorem (Theorem 3.2) 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, $[\![t]\!]^{\mathcal{M}} : A \to \mathcal{M}\, B$.

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

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.

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

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 $\mathsf{witness}$ expressions lies in their commitment Merkle root.

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

where $\mathrm{tag}^c_{\mathsf{witness}}$ value is derived as the SHA-256 hash of a unique value.

$$\mathrm{tag}^c_{\mathsf{witness}} \ := \ \mathrm{SHA256}[\texttt{53696d706c69636974791f436f6d6d69746d656e741f7769746e657373}]$$

Notice that a $\mathsf{witness}\, b$ expression does not commit to its parameter in the commitment root. This means that at redemption time a $\mathsf{witness}$ expression's parameter, called a *witness value*, could be set to any value.

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, $\mathsf{witness}$ expressions lets input data appear right where it is needed.

4.2.1 Elided Computation

4.2.2 Witness Merkle Root

4.2.3 Type Inference with Witness

Like other expressions, a $\mathsf{witness}$ 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:

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

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 to support the new assertion and fail expressions

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

where $\mathrm{tag}^{c}_{\mathsf{fail}}$ value is derived as the SHA-256 hash of a unique value.

$$\mathrm{tag}^{c}_{\mathsf{fail}} := \mathrm{SHA256}\,\mathtt{[53696d706c69636974791f436f6d6d69746d656e741f6661696c]}$$

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,

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

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.

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

$$\begin{aligned}
\mathsf{salted}^0\, h t &:= \mathsf{witness}\, 0_{\mathbb{2}} \vartriangle \mathsf{iden}; \mathsf{assertl}\, (\mathsf{drop}\, t)\, \#^c(\mathsf{fail}\, h) : A \vdash B \\
\mathsf{salted}^1\, h t &:= \mathsf{witness}\, 1_{\mathbb{2}} \vartriangle \mathsf{iden}; \mathsf{assertr}\, \#^c(\mathsf{fail}\, h)\, (\mathsf{drop}\, t) : A \vdash B
\end{aligned}$$

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 this document we will be detailing the primitives used for Bitcoin, or a Bitcoin-like application.

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 context, called BCEnv.

$$\begin{aligned}
\mathrm{Lock} &:= \mathbb{2}^{32} \\
\mathrm{Value} &:= \mathbb{2}^{64} \\
\mathrm{Outpoint} &:= \mathbb{2}^{256} \times \mathbb{2}^{32} \\
\mathrm{SigInput} &:= \left\{ \begin{array}{l} \mathrm{prevOutpoint} : \mathrm{Outpoint} \\ \mathrm{value} : \mathrm{Value} \\ \mathrm{sequence} : \mathbb{2}^{32} \end{array} \right\} \\
\mathrm{SigOutput} &:= \left\{ \begin{array}{l} \mathrm{value} : \mathrm{Value} \\ \mathrm{pubScript} : (\mathbb{2}^{8})^{*} \end{array} \right\} \\
\mathrm{SigTx} &:= \left\{ \begin{array}{l} \mathrm{version} : \mathbb{2}^{32} \\ \mathrm{inputs} : \mathrm{SigInput}^{+} \\ \mathrm{outputs} : \mathrm{SigOutput}^{+} \\ \mathrm{lockTime} : \mathrm{Lock} \end{array} \right\} \\
\mathrm{BCEnv} &:= \left\{ \begin{array}{l} \mathrm{tx} : \mathrm{SigTx} \\ \mathrm{ix} : \mathbb{2}^{32} \\ \mathrm{scriptCMR} : \mathbb{2}^{256} \end{array} \right\}
\end{aligned}$$

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 strictly less than 2^{32} by Bitcoin's deserialization implementation. Similarly, the length of SigOutput's pubScript is limited to strictly less than 2^{32} bytes. We assume all transactions to adhere to these limits when reasoning about Bitcoin transactions.

Furthermore, we assume that for every $e : \mathrm{BCEnv}$ that $\lceil e[\mathrm{ix}] \rceil < |e[\mathrm{tx}][\mathrm{inputs}]|$ so that "current" index being validated is, in fact, an input of the transaction.

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.

$$0 \leq \mathrm{fold}_{\mathbb{N}}^{\langle +,0\rangle}((\lambda o.\, \lceil o[\mathrm{value}] \rceil_{64})^{+}(e[\mathrm{tx}][\mathrm{outputs}])) \leq \mathrm{fold}_{\mathbb{N}}^{\langle +,0\rangle}((\lambda i.\, \lceil i[\mathrm{value}] \rceil_{64})^{+}(e[\mathrm{tx}][\mathrm{inputs}])) \leq 21000000 \times 10^8$$

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.

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

BC is a commutative, idempotent monad with zero:

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

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

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

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

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

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

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

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

$$\frac{}{\mathsf{currentPrevOutpoint} : \mathbb{1} \vdash \mathrm{OutPoint}}$$

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

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

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

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

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

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

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

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

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

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

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

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{inputsHash}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ (\eta^{\mathrm{S}} \circ \mathrm{SHA256} \circ \mu^{*} \circ \eta^{S} \circ \mathrm{inputHash}^{+})(e[\mathrm{tx}][\mathrm{inputs}])\\
[\![\mathsf{outputsHash}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ (\eta^{\mathrm{S}} \circ \mathrm{SHA256} \circ \mu^{*} \circ \eta^{S} \circ \mathrm{outputHash}^{+})(e[\mathrm{tx}][\mathrm{outputs}])\\
[\![\mathsf{numInputs}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}\lfloor |e[\mathrm{tx}][\mathrm{inputs}]| \rfloor_{32}\\
[\![\mathsf{totalInputValue}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}\big(\mathrm{fold}^{\lfloor + \rfloor_{64}}((\lambda l.\, l[\mathrm{value}])^{+}(e[\mathrm{tx}][\mathrm{inputs}]))\big)\\
[\![\mathsf{currentPrevOutpoint}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \mathrm{S}(\lambda l.\, l[\mathrm{prevOutpoint}])(e[\mathrm{tx}][\mathrm{inputs}]\lceil e[\mathrm{ix}]\rceil)\\
[\![\mathsf{currentValue}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \mathrm{S}(\lambda l.\, l[\mathrm{value}])(e[\mathrm{tx}][\mathrm{inputs}]\lceil e[\mathrm{ix}]\rceil)\\
[\![\mathsf{currentSequence}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \mathrm{S}(\lambda l.\, l[\mathrm{sequence}])(e[\mathrm{tx}][\mathrm{inputs}]\lceil e[\mathrm{ix}]\rceil)\\
[\![\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{numOutputs}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}\lfloor |e[\mathrm{tx}][\mathrm{outputs}]| \rfloor_{32}\\
[\![\mathsf{totalOutputValue}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}\big(\mathrm{fold}^{\lfloor + \rfloor_{64}}((\lambda l.\, l[\mathrm{value}])^{+}(e[\mathrm{tx}][\mathrm{outputs}]))\big)\\
[\![\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{pubScript}]))(e[\mathrm{tx}][\mathrm{outputs}]\lceil i\rceil))\\
[\![\mathsf{scriptCMR}]\!]^{\mathrm{BC}}\langle\rangle &:= \lambda e : \mathrm{BCEnv}.\ \eta^{\mathrm{S}}(e[\mathrm{scriptCMR}])
\end{aligned}$$

where

$$\begin{aligned}
\mathrm{inputHash}(l) &:= \mathrm{BE}_{256}(\pi_1(l[\mathrm{prevOutpoint}]))\cdot\mathrm{LE}_{32}(\pi_2(l[\mathrm{prevOutpoint}]))\cdot\mathrm{LE}_{32}(l[\mathrm{sequence}])\\
\mathrm{ouputHash}(l) &:= \mathrm{LE}_{64}(l[\mathrm{value}])\cdot\mathrm{BE}_{256}(\mathrm{SHA256}(l[\mathrm{pubScript}]))
\end{aligned}$$

Consider making everything big endian?

For most of these primitive expressions, it is clear that they can never fail, in the sense of never returning $\emptyset^{\mathrm{S}}$. The expressions $[\![\mathsf{currentPrevOutpoint}]\!]^{\mathrm{BC}}$, $[\![\mathsf{currentValue}]\!]^{\mathrm{BC}}$ and $[\![\mathsf{currentSequence}]\!]^{\mathrm{BC}}$ look like they could fail under some circumstances; however the assumption that for every $e : \mathrm{BCEnv}$ that $\lceil e[\mathrm{ix}]\rceil < |e[\mathrm{tx}][\mathrm{inputs}]|$ implies that those expressions cannot fail either.

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 to support the new expressions by hashing new unique byte strings.

$$\begin{aligned}
\#^c(\mathsf{version}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[76657273696f6e]})\\
\#^c(\mathsf{lockTime}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[6c6f636b54696d65]})\\
\#^c(\mathsf{inputsHash}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[696e7075747348617368]})\\
\#^c(\mathsf{outputsHash}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[6f75747075747348617368]})\\
\#^c(\mathsf{numInputs}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[6e756d496e70757473]})\\
\#^c(\mathsf{totalInputValue}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[746f74616c496e70757456616c7565]})\\
\#^c(\mathsf{currentPrevOutpoint}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[63757272656e74507265764f7574706f696e74]})\\
\#^c(\mathsf{currentValue}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[63757272656e7456616c7565]})\\
\#^c(\mathsf{currentSequence}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[63757272656e7453657175656e6365]})\\
\#^c(\mathsf{currentIndex}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[63757272656e74496e646578]})\\
\#^c(\mathsf{inputPrevOutpoint}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[696e707574507265764f7574706f696e74]})\\
\#^c(\mathsf{inputValue}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[696e70757456616c7565]})\\
\#^c(\mathsf{inputSequence}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[696e70757453657175656e6365]})\\
\#^c(\mathsf{numOutputs}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[6e756d784f757470757473]})\\
\#^c(\mathsf{totalOutputValue}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[746f74616c4f757470757456616c7565]})\\
\#^c(\mathsf{outputValue}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[6f757470757456616c7565]})\\
\#^c(\mathsf{outputScriptHash}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[6f757470757453637269707448617368]})\\
\#^c(\mathsf{scriptCMR}) &:= \mathrm{SHA256}(\mathrm{BCprefix}\cdot\texttt{[736372697074434d52]})
\end{aligned}$$

where

$$\mathrm{BCprefix} := \texttt{[53696d706c69636974791f5072696d69746976651f426974636f696e1f]}$$

4.5 Simplicity Programs

Ultimately, we only are interested in the side-effects of Simplicity expression; 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 $\mathsf{witness}$ 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 $[\![t]\!]^{\mathrm{BC}}\langle\rangle(e)=1_{\mathbb{2}}$, and all inputs must authorize the transaction if the transaction is valid.

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

4.5.1 Example: $\mathsf{checkSigHashAll}$

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 $\mathsf{schnorrVerify}$ called $\mathsf{schnorrAssert}$:

$$\mathsf{schnorrAssert} := \mathsf{assert}\ \mathsf{schnorrVerify} : (\mathrm{PubKey}\times\mathrm{Msg})\times\mathrm{Sig}\vdash\mathbb{1}$$

such that

$$[\![\mathsf{schnorrAssert}]\!]^{\mathrm{S}}\langle\langle p, m\rangle, s\rangle = \eta^{\mathrm{S}}\langle\rangle \;\Leftrightarrow\; [\![\mathsf{schnorrVerify}]\!]\langle\langle p, m\rangle, s\rangle = 1_2.$$

Next, we use the Bitcoin transaction extension to build a sigHashAll 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.

$$\begin{aligned}
&\mathsf{sigAll} : \mathbb{1} \vdash 2^{512} \times 2^{512}\\
&\mathsf{sigAll} \;:=\; (\mathsf{inputsHash} \vartriangle \mathsf{outputsHash})\\
&\qquad \vartriangle\; (((\mathsf{currentValue} \vartriangle (\mathsf{currentIndex} \vartriangle \mathsf{lockTime}))\\
&\qquad \vartriangle\; ((\mathsf{version} \vartriangle \mathsf{scribe}(\lfloor 2^{31}\rfloor_{32})) \vartriangle \mathsf{scribe}(\lfloor 0\rfloor_{64}))) \vartriangle \mathsf{scribe}(\lfloor 1184\rfloor_{256}))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{ivSigAll} : \mathbb{1} \vdash 2^{256}\\
&\mathsf{ivSigAll} \;:=\; (\mathsf{scribe} \circ \mathrm{SHA256})(\texttt{[53696d706c69636974791f5369676e61747572651d]} \cdot \mathrm{BE}_{256}(\#^c(\mathsf{sigAll})))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{sigHashAll} : \mathbb{1} \vdash \mathrm{Msg}\\
&\mathsf{sigHashAll} \;:=\; \mathsf{sigAll}; (\mathsf{ivSigAll} \vartriangle \mathsf{OH}; \mathsf{sha256\text{-}block}) \vartriangle \mathsf{IH}; \mathsf{sha256\text{-}block}
\end{aligned}$$

The sigAll expression reads a total of 672 bits of data from the environment. The ivSigAll 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.

Notice that in ivSigAll, in addition hashing the transaction data itself, we also hash the semantics of the data by including the commitment Merkle root of the Simplicity expression that generates the data. This fulfills the same role that including Bitcoin's sighash flags in its signed data does.

Finally, we combine everything together into a single Simplicity program. Given a public key in compressed format $p : \mathrm{PubKey}$ and a formatted Schnorr signature $s : \mathrm{Sig}$ we can create the Simplicity program $\mathsf{checkSigHashAll}\langle p, s\rangle : \mathbb{1} \vdash \mathbb{1}$

$$\mathsf{checkSigHashAll}\langle p, s\rangle := (\mathsf{scribe}_{\mathrm{PubKey}}(p) \vartriangle \mathsf{sigHashAll}) \vartriangle \mathsf{witness}_{\mathrm{Sig}}(s); \mathsf{schnorrAssert}$$

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 knowing 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 input number.

4.6 Schnorr Signature Aggregation

4.7 Malleability

4.7.1 Transaction Weight

Chapter 5
Jets

Reminder: build jets for parsing $\mathsf{lockTimes}$.

Chapter 6
Delegation

Our last Simplicity extension is the disconnect expression. 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*.

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

Semantically, the disconnect expression behaves similar to the composition expression, but where the commitment Merkle root of the expression t is passed as an argument to the expression s. We extend the formal semantics to the disconnect expression as follows.

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

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.

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

where the $\mathrm{tag}^c_{\mathsf{disconnect}}$ value is derived as the SHA-256 hash of a unique value.

$$\mathrm{tag}^c_{\mathsf{disconnect}} \;:=\; \mathrm{SHA256}[\mathtt{53696d706c69636974791f436f6d6d69746d656e741f646973636f6e6e656374}]$$

The commitment Merkle root only commits to the first argument, s, of a disconnect expression. During redemption the second argument, t, can be freely set to any Simplicity expression. In order to place restrictions on what t is allowed to be, 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, but still maintaining full control, because any redemption requires their signature on t. For example, a user can have t return 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 expression 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 could, a priori, perform arbitrarily much computation. Indeed we can use disconnect to commit to what are effectively unbounded loops in Simplicity (see Section 6.1). Contrast this with the witness expression, where type inference limits the size of the witness data, so such 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 Unbounded Loops

Chapter 7
Type Inference and Serialization

In this chapter we will define a representation of Simplicity expressions as an untyped Simplicity DAG where subexpressions are 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. An explicit Simplicity DAG is a topological sorted list of Simplicity nodes. Each Simplicity node is a combinator name with by 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.

$$\frac{}{\text{‘}\mathsf{iden}\text{’} : \mathrm{Node}}$$

$$\frac{}{\text{‘}\mathsf{unit}\text{’} : \mathrm{Node}}$$

$$\frac{i : \mathbb{N}}{\text{‘}\mathsf{injl}\text{’}\, i : \mathrm{Node}}$$

$$\frac{i : \mathbb{N}}{\text{‘}\mathsf{injr}\text{’}\, i : \mathrm{Node}}$$

$$\frac{i : \mathbb{N}}{\text{‘}\mathsf{take}\text{’}\, i : \mathrm{Node}}$$

$$\frac{i : \mathbb{N}}{\text{‘}\mathsf{drop}\text{’}\, i : \mathrm{Node}}$$

$$\frac{i : \mathbb{N} \qquad j : \mathbb{N}}{\text{‘}\mathsf{comp}\text{’}\, i\, j : \mathrm{Node}}$$

$$\frac{i : \mathbb{N} \qquad j : \mathbb{N}}{\text{‘}\mathsf{case}\text{’}\, i\, j : \mathrm{Node}}$$

$$\frac{i : \mathbb{N} \qquad j : \mathbb{N}}{\text{‘}\mathsf{pair}\text{’}\, i\, j : \mathrm{Node}}$$

$$\frac{i : \mathbb{N} \qquad j : \mathbb{N}}{\text{‘}\mathsf{disconnect}\text{’}\, i\, j : \mathrm{Node}}$$

$$\frac{v : \mathbb{2}^*}{\text{‘}\mathsf{witness}\text{’}\, v : \mathrm{Node}}$$

$$\frac{b : \mathbb{2}^{512}}{\text{‘}\mathsf{fail}\text{’}\, b : \mathrm{Node}}$$

$$\frac{h : \mathbb{2}^{256}}{\text{‘}\mathsf{hidden}\text{’}\, h : \mathrm{Node}}$$

In addition to the above every primitive name is a Node. This set of primitives is application specific but 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.

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

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

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

A list l: DAG must satisfy a condition that references only have offset that refer to nodes occurring strictly earlier in the list in order to be a well-formed DAG:

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

7.1.1 Type Inference

Simplicity DAGs, as described above, do not have type information associated with them. Before we can interpret DAGs as Simplicity expressions 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 list 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{array}{rcll}
\mathrm{con}\langle k, \text{‘iden’}\rangle & := & \{\alpha_k \doteq \beta_k\} & \\
\mathrm{con}\langle k, \text{‘unit’}\rangle & := & \{\beta_k \doteq \mathbb{1}\} & \\
\mathrm{con}\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} \\
\mathrm{con}\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} \\
\mathrm{con}\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} \\
\mathrm{con}\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} \\
\mathrm{con}\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}\} & \\
\mathrm{con}\langle k, \text{‘case’}\, i\, j\rangle & := & \{\alpha_k \doteq (\gamma_1 + \gamma_2) \times \gamma_3, \alpha_{k-i} \doteq \gamma_1 \times \gamma_3, a_{k-j} \doteq \gamma_2 \times \gamma_3, \beta_k \doteq \beta_{k-i} \doteq \beta_{k-j}\} & \\
 & & & \text{where } \gamma_1, \gamma_2, \gamma_3 \text{ are fresh} \\
\mathrm{con}\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}\} & \\
\mathrm{con}\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} \\
\mathrm{con}\langle k, \text{‘witness’}\, v\rangle & := & \{\} & \\
\mathrm{con}\langle k, \text{‘fail’}\, b\rangle & := & \{\} & \\
\mathrm{con}\langle k, \text{‘hidden’}\, h\rangle & := & \{\} & \\
\mathrm{con}\langle k, \text{‘version’}\rangle & := & \{\alpha_k \doteq \mathbb{1}, \beta_k \doteq 2^{32}\} & \\
\mathrm{con}\langle k, \text{‘lockTime’}\rangle & := & \{\alpha_k \doteq \mathbb{1}, \beta_k \doteq \mathrm{Lock}\} & \\
\vdots & & \vdots &
\end{array}$$

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.

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

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

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 $\varsigma(\alpha)$:

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

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 need 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 [15], 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, from computing witness Merkle roots to evaluation of Simplicity expressions using the Bit Machine, 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 should 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 [10], however because removing duplicated sub-DAGs can produce exponentially smaller terms it could be 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.

TODO: something about type-inference in the presence of sharing of hidden nodes? It might cause sharing that would otherwise be impossible with assertl and assertr?

7.1.2 Reconstructing a Simplicity Expressions

Given a 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

$$\begin{array}{rcll}
\mathrm{syn}(l, k, \emptyset^{\mathrm{S}}) & := & \bot & \\
\mathrm{syn}(l, k, \eta^{\mathrm{S}}(\text{'iden'})) & := & \mathsf{iden}_{\varsigma_{\mathbb{1}}(\alpha_k)} & \\
\mathrm{syn}(l, k, \eta^{\mathrm{S}}(\text{'unit'})) & := & \mathsf{unit}_{\varsigma_{\mathbb{1}}(\beta_k)} & \\
\mathrm{syn}(l, k, \eta^{\mathrm{S}}(\text{'injl'}\ i)) & := & \mathsf{injl}_{\varsigma_{\mathbb{1}}(\alpha_k), B, C}\ \mathrm{syn}(l, k - i, l[k - i]) & \text{where } B + C = \varsigma_{\mathbb{1}}(\beta_k)
\end{array}$$

$$
\begin{array}{rcll}
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{‘}\mathsf{injr}\text{’}\,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{‘}\mathsf{take}\text{’}\,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{‘}\mathsf{drop}\text{’}\,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{‘}\mathsf{comp}\text{’}\,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{‘}\mathsf{case}\text{’}\,i\,j)) & := & \mathrm{syncase}(l,k,k-i,l[k-i],k-j,l[k-j]) & \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{‘}\mathsf{pair}\text{’}\,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{‘}\mathsf{disconnect}\text{’}\,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{‘}\mathsf{witness}\text{’}\,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{‘}\mathsf{fail}\text{’}\,b)) & := & \mathsf{fail}_{\varsigma_{\mathbb{1}}(\alpha_k),\varsigma_{\mathbb{1}}(\beta_k)}\,b & \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{‘}\mathsf{hidden}\text{’}\,h)) & := & \bot & \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{‘}\mathsf{version}\text{’})) & := & \mathsf{version} & \\
\mathrm{syn}(l,k,\eta^{\mathrm{S}}(\text{‘}\mathsf{lockTime}\text{’})) & := & \mathsf{lockTime} & \\
\vdots & & \vdots &
\end{array}
$$

The case and witness clauses require special consideration which we define below. The syn function fails if it encounters a ‘hidden’ node. 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 we do not allow both branches of a case expression to be hidden.

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

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.

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

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

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 syn(k, '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 A describes an alternative, byte-based, binary encoding.

7.2.1 Serializtion of Bit String 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 $n : \mathbb{N}$ 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.

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

For example,

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

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 $\lfloor n \rfloor_{2^*}$.

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

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

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

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:

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

Below we give codes for Bitcoin primitive names, but each Simplicity application will have its own set of codes for its own primitives. The prefix codes for primitive names all begin with $\texttt{[10]}_2$.

$$\begin{aligned}
\ulcorner\text{‘version’}\urcorner &= \texttt{[1000000]}_2\\
\ulcorner\text{‘lockTime’}\urcorner &= \texttt{[1000001]}_2\\
\ulcorner\text{‘inputsHash’}\urcorner &= \texttt{[100001]}_2\\
\ulcorner\text{‘outputsHash’}\urcorner &= \texttt{[100010]}_2\\
\ulcorner\text{‘numInputs’}\urcorner &= \texttt{[100011]}_2\\
\ulcorner\text{‘totalInputValue’}\urcorner &= \texttt{[100100]}_2\\
\ulcorner\text{‘currentPrevOutpoint’}\urcorner &= \texttt{[100101]}_2\\
\ulcorner\text{‘currentValue’}\urcorner &= \texttt{[100110]}_2\\
\ulcorner\text{‘currentSequence’}\urcorner &= \texttt{[100111]}_2\\
\ulcorner\text{‘currentIndex’}\urcorner &= \texttt{[1010000]}_2\\
\ulcorner\text{‘inputPrevOutpoint’}\urcorner &= \texttt{[1010001]}_2\\
\ulcorner\text{‘inputValue’}\urcorner &= \texttt{[101001]}_2\\
\ulcorner\text{‘inputSequence’}\urcorner &= \texttt{[101010]}_2\\
\ulcorner\text{‘numOutputs’}\urcorner &= \texttt{[101011]}_2\\
\ulcorner\text{‘totalOutputValue’}\urcorner &= \texttt{[101100]}_2\\
\ulcorner\text{‘outputValue’}\urcorner &= \texttt{[101101]}_2\\
\ulcorner\text{‘outputScriptHash’}\urcorner &= \texttt{[101110]}_2\\
\ulcorner\text{‘scriptCMR’}\urcorner &= \texttt{[101111]}_2
\end{aligned}$$

TODO: [11] prefix reserved for jets

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

$$\text{stopCode}(l) := (\lambda x.\ulcorner x\urcorner)^{+}(l)\cdot\texttt{[01011]}_2$$

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.

$$\text{lengthCode}(l) := \ulcorner|l|\urcorner\cdot(\lambda x.\ulcorner x\urcorner)^{+}(l)$$

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 $(\lambda x.\ulcorner x\urcorner)^{+}(l)$. 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.

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 [16], 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 3.2) 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 parameteric 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

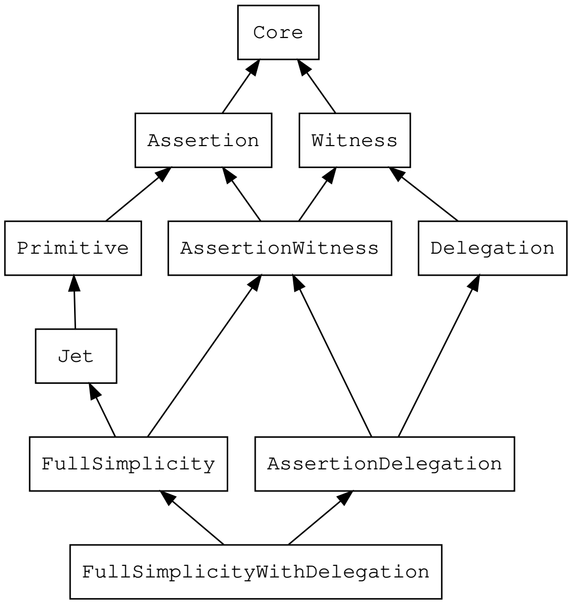

Figure 8.1. The inheritance hierarchy of algebras for Simplicity's partial extensions in Coq.

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" [9] and "Packaging Mathematical Structures" [6].

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 `Jets` 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 `Primtiive` 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 witness 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[1] 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

$$[\Theta \triangleright r_0 \cdot \texttt{[}\underline{c_1} \cdots\ c_{n_0}\texttt{]} \cdot r_0' | w_0 \cdot \texttt{[}c_1\ \cdots\ c_{n_1}\underline{\texttt{]}}\,\texttt{[?]}^{n_2}\texttt{[?]}^{m} \triangleleft \Xi]$$

where we are locally interested in what is immediately in front of the active read frame's cursor, $\texttt{[}\underline{c_1} \cdots\ c_{n_0}\texttt{]}$, and what is immediately surrounding the active write frame's cursor, $\texttt{[}c_1\ \cdots\ c_{n_1}\underline{\texttt{]}}\,\texttt{[?]}^{n_2}$. 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 $[\Theta \triangleright r_0 \cdot \bullet \cdot r_0' | w_0 \cdot \bullet \cdot \texttt{[?]}^{m} \triangleleft \Xi]$ 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 `Contexts` 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 $S_1 \rightarrowtail k \twoheadrightarrow S_2$ 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 $\text{bitSize}(A)$ and $\ulcorner a \urcorner_A$ respectively.

The Naive.translate structure provides a Simplicity algebra for Bit Machine Progams 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 development for Simplicity is found in the `Haskell` directory. The `Haskell/Tests.hs` file imports the various test modules throughout the development to build a testing executable to run them all.

9.1 Simplicity Types

The `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 is 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 asgg 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.

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 $\mathbb{2}$, 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. 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 popular word sizes between 1 and 512 bits.

9.2 Simplicity Terms

Terms are represented in tagless-final style [3]. This style is analogous to the "final" representation of terms that is defined in the Coq library. The development of the term language for full Simplicity is split into two files.

The `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 `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 `(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 `Simplicity/Semantics.hs` and it is discussed in Section 9.5.

The `Simplicity/Term.hs` module provides the blockchain primitives and jet extensions, in addition to re-exporting the `Simplicity/Term/Core.hs` module. This separation lets `Simplicity/Term/Core.hs` remain independent of the blockchain specific `Simplicity/Primitive.hs` module. 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 9.5.

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 whenh 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.

9.3 Blockchain Primitives

We aim to keep the Haskell library of Simplicity modular over different blockchain applications. Different blockchain applications are provided in the `Simplicity/Primitive` directory. At the moment only the Bitcoin blockchain application is provided by the `Simplicity/Primitive/Bitcoin.hs`.

The `Simplicity/Primitive.hs` module provides an interface to the different possible primitives provided by different blockchain applications. This module exports

- `Prim a b`, a GADT for different primitive expressions,
- `primPrefix` and `primName` which are used to generate unique names for the Merkle roots of primitive expressions,
- `PrimEnv` and `primSem`, which provides the type of the context and the denotational semantics for evaluating primitive expressions.

The library, by default, re-exports these values from the `Simplicity/Primitive/Bitcoin.hs` module. For other blockchain applications, one can modify the file to re-export the other application's module for primitives. The Kleisli morphisms over a reader monad over `PrimEnv` supports the semantics of primitive expressions.

9.3.1 Bitcoin Primitives

The `Simplicity/Primitive/Bitcoin.hs` module provides 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 `Simplicity/Primitive/Bitcoin/DataTypes.hs` module provides the data structures that make up the signed transaction data for Bitcoin.

9.4 Merkle Roots

The `Simplicity/MerkleRoot.hs` module provides instances of Simplicity terms that compute the commitment and witness Merkle roots. The `commitmentRoot` and `witnessRoot` 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 `Simplicity/Digest.hs`.

9.5 Denotational Semantics of Full Simplicity

The `Simplicity/Term.hs` module provides `(->)` and `Kleisli m` instances for the full Simplicity language excluding delegation. Semantics for the full Simplicity language with delegation, which depends on computing commitment Merkle roots, is found in the `Simplicity/Semantics.hs` module.

The `Delegator p a b` helper type bundles a commitment Merkle root computation with the regular Simplicity semantics, 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 an instance of `Delegator` for the Kleisli semantics that support the Blockchain primitives, and thus is an instance of the full Simplicity language with delegation. The `sem` function unwraps all the type wrappers of `Semantics a b` and provides a concrete function from `PrimEnv` and `a` to `Maybe b`.

9.6 Example Simplicity Expressions

The `Simplicity/Programs` directory contains various developments of Simplicity expressions in Haskell. The `Simplicity/Programs/Tests.hs` has some QuickCheck properties that provide randomized testing for some of the programs defined in this section.

9.6.1 Bits

The `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.

9.6.2 Multi-bit Words

The `Simplicity/Programs/Word.hs` file provides support for multi-bit word expressions that operate on Simplicity's word types.

9.6.2.1 Arithmetic operations

The `Simplicity/Programs/Word.hs` file provides the standard implementations of the `zero`, `adder`, `fullAdder`, `subtractor`, `fullSubtractor`, `multiplier`, and `fullMultiplier` Simplicity expressions. Notice that the implementation of these functions is careful to use explicit sharing of Simplicity sub-expressions where possible through the `where` clauses.

9.6.2.2 Bit-wise operations

The `shift` and `rotate` functions create Simplicity expressions that do right shifts and rotates of multi-bit words by any constant amount. Left (unsigned) shifts and rotates can be made by passing a negative value for the shift/rotate amount.

The `bitwise` 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 bit-wise. There is also a variant, called `bitwiseTri` the does the same thing for trinary bit operations.

9.6.3 Generic

The `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.

9.6.4 SHA-256

The `Simplicity/Programs/Sha256.hs` file provides Simplicity expressions to help compute SHA-256 hashes. The `iv` Simplicity expression is a constant function the returns the initial value to begin a SHA-256 computation. The `hashBlock` Simplicity 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.

9.6.5 LibSecp256k1

The `Simplicity/Programs/LibSecp256k1.hs` file provides Simplicity expressions that mimic the functional behaviour of the the libsecp256k1 elliptic curve library [18]. This includes Simplicity types for, and operations on secp256k1's underlying finite field with the `10x26` limb representation, elliptic curve point operations in affine and Jacobian coordinates, and linear combinations of points.

This module also include the `schnorrVerify` and `schnorrAssert` expressions that implement Schnorr signatures as specified in BIP-Schnorr [17].

9.6.6 CheckSigHashAll

The `Simplicity/Programs/CheckSigHashAll.hs` file provides the `checkSigHashAll` Simplicity expression that verifies Schnorr signature over the transaction data hash produced by `sigHashAll` for a provided public key. Some variants of this expression are also provided including `pkwCheckSigHashAll` which builds a complete Simplicity program from a given public key and signature.

9.7 The Bit Machine

The `Simplicity/BitMachine/` directory has modules related to the Bit Machine and evaluation of Simplicity via the Bit Machine.

The `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 `Simplicity/BitMachine.hs` file (technically not in the `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 `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.

9.7.1 Translating Simplicity to the Bit Machine

The `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.

The `Simplicity/BitMachine/Tests.hs` runs a few of the example 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.

9.7.2 Static Analysis

The `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 `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 `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.

9.8 Type Inference

The file `Simplicity/Inference.hs` defines a concrete term data type for Simplicity expressions in open recursive style via the `TermF ty` 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. When type annotations are unused, this `ty` parameter is set to `()`, as is the case for the `UntypedTermF` functor synonym.

While the `Data.Functor.Fixedpoint.Fix` of this `TermF ty` functor would yield a type for untyped, full Simplicity terms, instead we usually use a list or vector of `TermF ty Integer` values to build a DAG structure, where the `Integer` values are references to other subexpressions withing the list or vector. This provides a structure with explicit sharing of subexpressions. This structure is captured by the `SimplicityDag` type synonym.

The `WitnessData` type is a specialized data structure to allowing the witness values to be pruned during encoding and decoding. The `getWitnessData` function restores the witness data as a value of a Simplicity type; however a the caller must know a suitable Simplicity type to interpret the witness data correctly.

The two principle functions of this module are the `typeInference` and `typeCheck` functions. The `typeInference` function discards the type annotations of the input Simplicity DAG and performs first-order unification to infer new, principle type annotations, with any remaining type variables instantiated at the `()` type. The `typeCheck` function performs the same unification, but also adds unification constraints for the input and output types of the expression as provided by the caller of the function. The type annotations are type checked and, if everything is successful, a proper well-typed Simplicity expression is returned.

9.9 Serialization

There are two main methods of serialization found in this Simplicity library. The first methods is via the `Get` and `PutM` monads from the `cereal` package. These are used for serializations to and from `ByteStrings`. The second 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 `Simplicity/Serializaiton.hs`.

9.9.1 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 equivalent 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 [14] 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 `Simplicity/Serializaiton.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.

9.9.2 Serialization of Simplicity DAGs

The file `Simplicity/Dag.hs` provides a `sortDag` 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. It also runs type inference to determine the principle type annotations needed to optimal sharing. The type inference is also used to prune away any unused witness data.

The file `Simplicity/Serialization/BitString.hs` provides `getDag` and `putDag` functions that decode and encode a Simplicity DAG structures, generated by `Simplicity.Dag.getDag`, as described in Section 7.2. The file `Simplicity/Serialization/ByteString.hs` provides the same functions for the encoding described in Appendix A.

Chapter 10
C Library Guide

Appendix A
Alternative Serialization of Simplicity DAGs

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.

$$\begin{aligned}
\mathrm{byteCode}(1,0) &:= \epsilon_{2^8} \\
\mathrm{byteCode}(b,i) &:= \mathrm{byteCode}(2^{8n},q)\cdot\mathrm{BE}(\lfloor i\rfloor_8) \qquad \text{where } 2^8\,q\le i<2^8(q+1) \text{ and } 2^{8n}<b\le 2^{8(n+1)} \text{ and } i<b
\end{aligned}$$

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.

$$\begin{aligned}
\mathrm{bytePad}(\epsilon) &:= \epsilon_{2^8} \\
\mathrm{bytePad}(b_0 \blacktriangleleft b_1 \blacktriangleleft b_2 \blacktriangleleft b_3 \blacktriangleleft b_4 \blacktriangleleft b_5 \blacktriangleleft b_6 \blacktriangleleft b_7 \blacktriangleleft v) &:= \langle\langle\langle b_0,b_1\rangle,\langle b_2,b_3\rangle\rangle,\langle\langle b_4,b_5\rangle,\langle b_6,b_7\rangle\rangle\rangle \blacktriangleleft \mathrm{bytePad}(v) \\
\mathrm{bytePad}(v) &:= \mathrm{bytePad}\big(v\cdot\texttt{[0]}^{8-|v|}\big) \qquad \text{when } 0<|v|<8
\end{aligned}$$

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, $\mathrm{byteCode}:\mathbb{N}\times\mathrm{Node}\to(2^8)^*$. 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}
\mathrm{byteCode}(k,\text{‘comp’}\,1\,1) &= \texttt{[00]}_{2^8} \\
\mathrm{byteCode}(k,\text{‘case’}\,1\,1) &= \texttt{[01]}_{2^8} \\
\mathrm{byteCode}(k,\text{‘pair’}\,1\,1) &= \texttt{[02]}_{2^8} \\
\mathrm{byteCode}(k,\text{‘disconnect’}\,1\,1) &= \texttt{[03]}_{2^8} \\
\mathrm{byteCode}(k,\text{‘comp’}\,1\,j) &= \texttt{[04]}_{2^8}\cdot\mathrm{byteCode}(k-1,j-2) && \text{where } 1<j \\
\mathrm{byteCode}(k,\text{‘case’}\,1\,j) &= \texttt{[05]}_{2^8}\cdot\mathrm{byteCode}(k-1,j-2) && \text{where } 1<j \\
\mathrm{byteCode}(k,\text{‘pair’}\,1\,j) &= \texttt{[06]}_{2^8}\cdot\mathrm{byteCode}(k-1,j-2) && \text{where } 1<j \\
\mathrm{byteCode}(k,\text{‘disconnect’}\,1\,j) &= \texttt{[07]}_{2^8}\cdot\mathrm{byteCode}(k-1,j-2) && \text{where } 1<j \\
\mathrm{byteCode}(k,\text{‘comp’}\,i\,1) &= \texttt{[08]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2) && \text{where } 1<i \\
\mathrm{byteCode}(k,\text{‘case’}\,i\,1) &= \texttt{[09]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2) && \text{where } 1<i \\
\mathrm{byteCode}(k,\text{‘pair’}\,i\,1) &= \texttt{[0a]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2) && \text{where } 1<i \\
\mathrm{byteCode}(k,\text{‘disconnect’}\,i\,1) &= \texttt{[0b]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2) && \text{where } 1<i \\
\mathrm{byteCode}(k,\text{‘comp’}\,i\,j) &= \texttt{[0c]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2)\cdot\mathrm{byteCode}(k-1,j-2) && \text{where } 1<i \text{ and } 1<j \\
\mathrm{byteCode}(k,\text{‘case’}\,i\,j) &= \texttt{[0d]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2)\cdot\mathrm{byteCode}(k-1,j-2) && \text{where } 1<i \text{ and } 1<j \\
\mathrm{byteCode}(k,\text{‘pair’}\,i\,j) &= \texttt{[0e]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2)\cdot\mathrm{byteCode}(k-1,j-2) && \text{where } 1<i \text{ and } 1<j \\
\mathrm{byteCode}(k,\text{‘disconnect’}\,i\,j) &= \texttt{[0f]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2)\cdot\mathrm{byteCode}(k-1,j-2) && \text{where } 1<i \text{ and } 1<j \\
\mathrm{byteCode}(k,\text{‘injl’}\,1) &= \texttt{[10]}_{2^8}
\end{aligned}$$

$$\begin{array}{rcll}
\mathrm{byteCode}(k,\text{`}\mathsf{injr}\text{'}\ 1) & = & \texttt{[11]}_{2^8} & \\
\mathrm{byteCode}(k,\text{`}\mathsf{take}\text{'}\ 1) & = & \texttt{[12]}_{2^8} & \\
\mathrm{byteCode}(k,\text{`}\mathsf{drop}\text{'}\ 1) & = & \texttt{[13]}_{2^8} & \\
\mathrm{byteCode}(k,\text{`}\mathsf{injl}\text{'}\ i) & = & \texttt{[18]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2) & \text{where } 1<i \\
\mathrm{byteCode}(k,\text{`}\mathsf{injr}\text{'}\ i) & = & \texttt{[19]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2) & \text{where } 1<i \\
\mathrm{byteCode}(k,\text{`}\mathsf{take}\text{'}\ i) & = & \texttt{[1a]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2) & \text{where } 1<i \\
\mathrm{byteCode}(k,\text{`}\mathsf{drop}\text{'}\ i) & = & \texttt{[1b]}_{2^8}\cdot\mathrm{byteCode}(k-1,i-2) & \text{where } 1<i \\
\mathrm{byteCode}(k,\text{`}\mathsf{iden}\text{'}) & = & \texttt{[20]}_{2^8} & \\
\mathrm{byteCode}(k,\text{`}\mathsf{unit}\text{'}) & = & \texttt{[21]}_{2^8} & \\
\mathrm{byteCode}(k,\text{`}\mathsf{fail}\text{'}\ b) & = & \texttt{[22]}_{2^8}\cdot\mathrm{BE}(b) & \\
\mathrm{byteCode}(k,\text{`}\mathsf{hidden}\text{'}\ h) & = & \texttt{[23]}_{2^8}\cdot\mathrm{BE}(h) & \\
\mathrm{byteCode}(k,\text{`}\mathsf{witness}\text{'}\ v) & = & \mathrm{BE}(\lfloor \mathsf{80}_{2^8}+|v|\rfloor_8)\cdot\mathrm{bytePad}(v) & \text{where } |v|<127 \\
\mathrm{byteCode}(k,\text{`}\mathsf{witness}\text{'}\ v) & = & \texttt{[ff]}_{2^8}\cdot\mathrm{BE}(\lfloor |v|\rfloor_{16})\cdot\mathrm{bytePad}(v) & \text{where } 127\leq|v|<2^{16}
\end{array}$$

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.

$$\begin{array}{rcl}
\mathrm{byteCode}(k,\text{`}\mathsf{version}\text{'}) & = & \texttt{[24]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{lockTime}\text{'}) & = & \texttt{[25]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{inputsHash}\text{'}) & = & \texttt{[26]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{outputsHash}\text{'}) & = & \texttt{[27]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{numInputs}\text{'}) & = & \texttt{[28]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{totalInputValue}\text{'}) & = & \texttt{[29]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{currentPrevOutpoint}\text{'}) & = & \texttt{[2a]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{currentValue}\text{'}) & = & \texttt{[2b]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{currentSequence}\text{'}) & = & \texttt{[2c]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{currentIndex}\text{'}) & = & \texttt{[2d]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{inputPrevOutpoint}\text{'}) & = & \texttt{[2e]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{inputValue}\text{'}) & = & \texttt{[2f]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{inputSequence}\text{'}) & = & \texttt{[30]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{numOutputs}\text{'}) & = & \texttt{[31]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{totalOutputValue}\text{'}) & = & \texttt{[32]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{outputValue}\text{'}) & = & \texttt{[33]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{outputScriptHash}\text{'}) & = & \texttt{[34]}_{2^8} \\
\mathrm{byteCode}(k,\text{`}\mathsf{scriptCMR}\text{'}) & = & \texttt{[35]}_{2^8}
\end{array}$$

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.

$$\mathrm{byteCode}(l) := ((\mu^* \circ \eta^{\mathrm{S}} \circ \mathrm{byteCode}^+ \circ \mathrm{indexed})(l))\cdot\texttt{[1f]}_{2^8}$$

Notice that while $\mathrm{byteCode}(0,x)$ for $x:\mathrm{Node}$ seems like it could call $\mathrm{byteCode}(-1,n)$ for some $n:\mathbb{N}$, this can never happen for Simplicity DAGs. Recall from Section 7.1, that a well-formed Simplicity DAG, $l:\mathrm{Node}^+$, satisfies the condition

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

The condition on DAGs implies that $|\mathrm{ref}(l[0])|=0$. This condition means that $\mathrm{byteCode}(-1,n)$ never occurs for any $n:\mathbb{N}$.

Bibliography

[1] A. W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7–1, apr 2015.

[2] Bitcoinwiki. Script. `https://en.bitcoin.it/w/index.php?title=Scriptoldid=61707` , 2016.

[3] J. Carette, O. Kiselyov and C. Shan. Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, sep 2009.

[4] Certicom Research. Standards for Efficient Cryptography 2: Recommended Elliptic Curve Domain Parameters. Standard SEC2, Certicom Corp., Mississauga, ON, USA, Sep 2000.

[5] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.7.* Oct 2017.

[6] F. Garillot, G. Gonthier, A. Mahboubi and L. Rideau. Packaging Mathematical Structures. In Tobias Nipkow and Christian Urban, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*. Munich, Germany, 2009. Springer.

[7] G. Gentzen. Investigations into logical deduction. In M.E. Szabo, editor, *The collected papers of Gerhard Gentzen*, Studies in logic and the foundations of mathematics, chapter 3. North-Holland Pub. Co., 1969.

[8] D. J. King and P. Wadler. *Combining Monads*, pages 134–143. Springer London, London, 1993.

[9] A. Mahboubi and E. Tassi. Canonical structures for the working Coq user. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, ITP'13, pages 19–34. Berlin, Heidelberg, 2013. Springer-Verlag.

[10] H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 382–401. New York, NY, USA, 1990. ACM.

[11] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, Nov 2008.

[12] S. Nakamoto. Re: Transactions and Scripts: DUP HASH160 ... EQUALVERIFY CHECKSIG. `https://bitcointalk.org/index.php?topic=195.msg1611#msg1611`, Jun 2010.

[13] National institute of standards and technology. FIPS 180-4, secure hash standard, federal information processing standard (FIPS), publication 180-4. Technical Report, DEPARTMENT OF COMMERCE, aug 2015.

[14] R. O'Connor. Van Laarhoven free monad. Feb 2014. Blog post, `http://r6.ca/blog/20140210T181244Z.html`.

[15] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[16] Wikipedia contributors. F-algebra — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=F-algebraoldid=814231684`, 2017.

[17] P. Wuille. Bip-schnorr. 2018. `Https://github.com/sipa/bips/blob/bip-schnorr/bip-schnorr.mediawiki`.

[18] P. Wuille. Libsecp256k1. `https://github.com/bitcoin-core/secp256k1/tree/1e6f1f5ad5e7f1e3ef79313ec02023902bf8175c`, May 2018.