# Simplicity: A New Language for Blockchains

Russell O'Connor
roconnor@blockstream.com

2017-10-30

**Abstract**

Simplicity is a typed, combinator-based, functional language without loops and recursion, designed to be used for crypto-currencies and blockchain applications. It aims to improve upon existing crypto-currency languages, such as Bitcoin Script and Ethereum's EVM, while avoiding some of the problems they face. Simplicity comes with formal denotational semantics defined in Coq, a popular, general purpose software proof assistant. Simplicity also includes operational semantics that are defined with an abstract machine that we call the Bit Machine. The Bit Machine is used as a tool for measuring the computational space and time resources needed to evaluate Simplicity programs. Owing to its Turing incompleteness, Simplicity is amenable to static analysis that can be used to derive upper bounds on the computational resources needed, prior to execution. While Turing incomplete, Simplicity can express any finitary function, which we believe is enough to build useful "smart contracts" for blockchain applications.

## 0 License




## 1 Introduction

Blockchain and distributed ledger technologies define protocols that allow large, ad-hoc, groups of users to transfer value between themselves without needing to trust each other or any central authority. Using public-key cryptography, users can sign transactions that transfer ownership of funds to other users. To prevent transactions from conflicting with each other, for example when one user attempts to transfer the same funds to multiple different users at the same time, a consistent sequence of blocks of transactions is committed using a proof of work scheme. This proof of work is created by participants called miners. Each user verifies every block of transactions; among multiple sequences of valid blocks, the sequence with the most proof of work is declared to be authoritative.

Bitcoin [20] was the first protocol to use this blockchain technology to create a secure and permissionless crypto-currency. It comes with a built-in programming language, called Bitcoin Script [6, 22], which determines if a transaction is authorized to transfer funds. Rather than sending funds to a specific party, users send funds to a specific Bitcoin Script program. This program guards the funds and only allows them to be redeemed by a transaction input that causes the program to return successfully. Every peer in the Bitcoin network must execute each of these programs with the input provided by the transaction data and all these programs must return successfully in order for the given transaction to be considered valid.

A typical program specifies a public key and simply requires that a valid digital signature of the transaction for that key be provided. Only someone with knowledge of the corresponding private key can create such a signature.

Funds can be guarded by more complex programs. Examples range from simpler programs that require signatures from multiple parties, permitting escrow services [14], to more complex programs that allow for zero-knowledge contingent payments [19], allowing for trustless and atomic purchases of digital data. This last example illustrates how Bitcoin Script can be used to build smart contracts, which allow parties to create conditional transactions that are enforced by the protocol itself.

Because of its Script language, Bitcoin is sometimes described as programmable money.

## 1.1 Bitcoin Script

Bitcoin Script is a stack-based language similar to Forth. A program in Bitcoin Script is a sequence of operations for its stack machine. Bitcoin Script has conditionals but no loops, thus all programs halt and the language is not Turing complete.

Originally, these programs were committed as part of the output field of a Bitcoin transaction. The program from the output being redeemed is executed starting with the initial stack provided by the input redeeming it. The program is successful if it completes without crashing and leaves a non-zero value on the top of stack.

Later, a new option called *pay to script hash* (P2SH) [1] was added to Bitcoin. Using this option, programs are committed by specifying a hash of the source code as part of the output field of a Bitcoin transaction. The input field of a redeeming Bitcoin transaction then provides both the program and the initial stack for the program. The transaction is valid if the hash of the provided program matches the hash specified in the output being redeemed, and if that program executes successfully with the provided input, as before. The main effect here is moving the cost of large programs from the person sending funds to the person redeeming funds.

Bitcoin Script has some drawbacks. Many operations were disabled by Bitcoin's creator, Satoshi Nakamoto [21]. This has left Bitcoin Script with a few arithmetic (multiplication was disabled), conditional, stack manipulation, hash-

ing, and digital-signature verification operations. In practice, almost all programs found in Bitcoin follow a small set of simple templates to perform digital signature verification.

However, Bitcoin Script also has some desirable properties. All Bitcoin Script operations are pure functions of the machine state expect for the signature-verification operations. These signature-verification operations require a hash of some of the transaction data. Together, this means the program's success or failure is purely a function of the transaction data. Therefore, the person creating a transaction can know whether the transaction they have created is valid or not.

Bitcoin Script is also amenable to static analysis, which is another desirable property. The digital-signature verification operations are the most expensive operations. Prior to execution, Bitcoin counts the number of occurrences of these operations to compute an upper bound on the number of expensive calls that may occur. Programs whose count exceeds a certain threshold are invalid. This is a form of static analysis that ensures the amount of computation that will be done isn't excessive. Moreover, before committing a program to an output, the creator of the program can ensure that the amount of computation that will be done will not be excessive.

## 1.2 Ethereum and the EVM

Ethereum [31] is another crypto-currency with programmable transactions. It defines a language called EVM for its transactions. While more expressive and flexible than Bitcoin Script, the design of Ethereum and the EVM has several issues.

The EVM is a Turing-complete programming language with a stack, random access memory, and persistent storage. To prevent infinite loops, execution is limited by a counter called gas, which is paid for in Ethereum's unit of account, Ether, to the miner of the block containing the transaction. Static analysis isn't practical with a Turing-complete language. Therefore, when a program runs out of gas, the transaction is nullified but the gas is still paid to the miner to ensure they are compensated for their computation efforts.

Because of direct access to persistent storage, EVM programs are a function of both the transaction and the state of the blockchain at the point the transaction is inserted into the blockchain. This means users cannot necessarily know what the result of their transaction will be when they create it, nor can they necessarily know how much gas will be consumed. Users must provide enough gas to cover the worst-case use scenario and there is no general purpose algorithm that can compute that bound. In particular, there is no practical general purpose way to tell if any given program will run out of gas in some context either at redemption time or creation time. There have been some efforts to perform static analysis on a subset of the EVM language [5], but these tools are limited (e.g. they do not support programs with loops).

Unlike Bitcoin, there are many instances of ad-hoc, one-off, special purpose programs in Ethereum. These programs are usually written in a language called

Solidity [12] and compiled to the EVM. These ad-hoc programs are regularly broken owing to the complex semantics of both Solidity and the EVM; the most famous of these failures were the DAO [8] and Parity's multiple signature validation program [26].

### 1.3 A New Language

In this paper, we propose a new language that maintains or enhances the desirable properties that Bitcoin Script has while adding expressiveness. Our design goals are:

- Create an expressive language that provides users with the tools needed to build novel programs and smart contracts.
- Enable static analysis that provides useful upper bounds on the amount of computation required.
- Minimize bandwidth and storage requirements and enhance privacy by removing unused code at redemption time.
- Maintain Bitcoin's design of self-contained transactions whereby programs do not have access to any information outside the transaction.
- Provide formal semantics that facilitate easy reasoning about programs using existing off-the-shelf proof-assistant software.

A few of these goals deserve additional explanation:

Static analysis allows the protocol to place limits on the amount of computation a transaction can have, so that nodes running the protocol are not overly burdened. Furthermore, the static analysis can provide program creators with a general purpose tool for verifying that the programs they build will always fit within these limits. Additionally, it is easy for the other participants in a contract to check the bounds on smart contract's programs themselves.

Self-contained transactions ensure that execution does not depend on the global state of the blockchain. Once a transaction's programs have been validated, that fact can be cached. This is particularly useful when transactions are being passed around the network before inclusion in the blockchain; once included in the blockchain, the programs do not need to be executed again.

Finally, formal semantics that work with proof-assistant software provide the opportunity for contract developers to reason about their programs to rule out logical errors and to help avoid scenarios like the DAO and Parity's multi-signature program failure.

Like Bitcoin Script and the EVM, our language is designed as low-level language for executing smart contracts, rather than as a language for programmers write directly. As such, we expect it to be a target for other, higher-level, languages to be complied to.

We call our new language *Simplicity*.

$$\frac{}{\mathsf{iden} : A \vdash A} \qquad \frac{s : A \vdash B \qquad t : B \vdash C}{\mathsf{comp}\, s\, t : A \vdash C}$$

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

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

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

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

Figure 1: Typing rules for the terms of core Simplicity.

# 2 Core Simplicity

Simplicity is a typed combinator language. Each well-typed Simplicity expression is associated with two types, an input type and an output type. Every expression denotes a function from its input type to its output type.

## 2.1 Types in Simplicity

Types in Simplicity come in three flavors.

- The unit type, written as $\mathbb{1}$, is the type with one element.
- A sum type, written as $A + B$, contains the tagged union of values from either the left type $A$ or the right type $B$.
- A product type, written as $A \times B$, contains pairs of elements with the first one from the type $A$ and the second one from the type $B$.

There are no recursive types in Simplicity. Every type in Simplicity only contains a finite number of values; the number of possible values a type contains can be computed by interpreting the type as an arithmetic expression.

## 2.2 Terms in Simplicity

The term language for Simplicity is based on Gentzen's sequent calculus [13]. We write a type-annotated Simplicity expression as $t : A \vdash B$ where $t$ a Simplicity expression, $A$ is the expression's input type, and $B$ is the expression's output type.

The core of Simplicity consists of nine combinators for building expressions. They capture the fundamental operations for the three flavors of types in Simplicity. The typing rules for these nine combinators is given in Figure 1.

We can classify the combinators based on the flavor of types they support.

- The unit term returns the singular value of the unit type and ignores its argument.
- The injl and injr combinators create tagged values, while the case combinator, Simplicity's branching operation, evaluates one of its two subexpressions based on the tag of the first component of its input.
- The pair combinator creates pairs, while the take and drop combinators access first and second components of a pair respectively.
- The iden and comp combinators are not specific to any flavor of type. The iden term represents the identity function for any type and the comp combinator provides function composition.

Simplicity expressions form an abstract syntax tree. The leaves of this tree are either iden or unit terms. The nodes are one of the other seven combinators. Each node has one or two children depending on which combinator the node represents.

Precise semantics are given in the next section. Extensions to this core set of combinators is given in Section 4.

### 2.3 Denotational Semantics of Simplicity

Before giving the semantics of Simplicity terms, we need notations for the values of Simplicity types.

- We write $\langle\rangle : \mathbb{1}$ for the singular value of the unit type.
- We write $\sigma^{\mathbf{L}}(a) : A + B$ when $a : A$ for left-tagged values of the sum type.
- We write $\sigma^{\mathbf{R}}(b) : A + B$ when $b : B$ for right-tagged values of the sum type.
- We write $\langle a, b\rangle : A \times B$ when $a : A$ and $b : B$ for values of the pair type.

We emphasize that these values are not directly representable in Simplicity because Simplicity expressions can only denote functions; we use these values only to define the functional semantics of Simplicity.

Simplicity's denotational semantics are recursively defined. For an expression $t : A \vdash B$, we define its semantics $[\![t]\!] : A \to B$ as follows.

$$
\begin{aligned}
[\![\mathsf{iden}]\!](a) &\coloneqq a \\
[\![\mathsf{comp}\, s\, t]\!](a) &\coloneqq [\![t]\!]([\![s]\!](a)) \\
[\![\mathsf{unit}]\!](a) &\coloneqq \langle\rangle \\
[\![\mathsf{injl}\, t]\!](a) &\coloneqq \sigma^{\mathbf{L}}([\![t]\!](a)) \\
[\![\mathsf{injr}\, t]\!](a) &\coloneqq \sigma^{\mathbf{R}}([\![t]\!](a)) \\
[\![\mathsf{case}\, s\, t]\!]\langle\sigma^{\mathbf{L}}(a), c\rangle &\coloneqq [\![s]\!]\langle a, c\rangle \\
[\![\mathsf{case}\, s\, t]\!]\langle\sigma^{\mathbf{R}}(b), c\rangle &\coloneqq [\![t]\!]\langle b, c\rangle \\
[\![\mathsf{pair}\, s\, t]\!](a) &\coloneqq \langle [\![s]\!](a), [\![t]\!](a)\rangle \\
[\![\mathsf{take}\, t]\!]\langle a, b\rangle &\coloneqq [\![t]\!](a) \\
[\![\mathsf{drop}\, t]\!]\langle a, b\rangle &\coloneqq [\![t]\!](b)
\end{aligned}
$$

We have formalized the language and semantics of core Simplicity in the Coq proof assistant [29]. The formal semantics in Coq are the official semantics of Simplicity, and for core Simplicity, it is short enough that it fits in Appendix A of this paper. Having semantics in a general purpose proof assistant allows us to formally reason both about programs written in Simplicity and about algorithms that analyze Simplicity programs.

## 2.4 Completeness

Simplicity cannot express general computation. It can only express finitary functions, because each Simplicity type contains only finitely many values. However, within this domain, Simplicity's set of combinators is complete: any function between Simplicity's types can be expressed.

**Theorem 2.1 (Finitary Completeness)** *Let $A$ and $B$ be Simplicity types. Let $f : A \to B$ be any function between these types. Then there exists some core Simplicity expression $t : A \vdash B$ such that $[\![t]\!] = f$.*

We have verified this theorem with Coq.

This theorem holds because functions between finite types can be described, in principle, by a lookup table that maps inputs to outputs. In principle, such a lookup table can be encoded as a Simplicity expression that does repeated case analysis on its input and returns a fixed-output value for each possible case.

However, using this theorem to construct Simplicity expressions is not practical. For example, a lookup table for the compression function for SHA-256 [24], which maps 768 bits to 256 bits, would be astronomical in size, requiring $2^{776}$ bits. To create practical programs in Simplicity, we need to take advantage of structured computation in order to succinctly express functions.

## 2.5 Example Programs

The combinators of core Simplicity may seem paltry, so in this section we illustrate how we can construct programs in Simplicity. We begin by defining a type for a bit, $2$, as the sum of two unit types.

$$2 := 1 + 1$$

We choose an interpretation of bits as numbers where we define the left-tagged value as denoting zero and the right tagged value as denoting one.

$$\lceil \sigma^{\mathbf{L}}\langle\rangle \rceil_2 := 0$$
$$\lceil \sigma^{\mathbf{R}}\langle\rangle \rceil_2 := 1$$

We can write Simplicity programs to manipulate bits. For example, we can define the $\mathsf{not}$ function to flip a bit.

$$\mathsf{not} : 2 \vdash 2$$
$$\mathsf{not} := \mathsf{comp}\,(\mathsf{pair}\,\mathsf{iden}\,\mathsf{unit})\,(\mathsf{case}\,(\mathsf{injr}\,\mathsf{unit})\,(\mathsf{injl}\,\mathsf{unit}))$$

By recursively taking products, we can define types for multi-bit words.

$$2^1 := 2$$
$$2^{2n} := 2^n \times 2^n$$

For example, the $2^{32}$ type can represent 32-bit words.

We recursively define the interpretation of pairs of words as numbers, choosing to use big-endian order.

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

We can write a half-adder of two bits in Simplicity.

$$\mathsf{half\text{-}adder} : 2 \times 2 \vdash 2^2$$
$$\mathsf{half\text{-}adder} := \mathsf{case}\ \ (\mathsf{drop}\,(\mathsf{pair}\,(\mathsf{injl}\,\mathsf{unit})\ \mathsf{iden}))$$
$$(\mathsf{drop}\,(\mathsf{pair}\,\mathsf{iden}\ \mathsf{not}))$$

We can prove the $\mathsf{half\text{-}adder}$ expression correct.

**Theorem 2.2 (Half Adder Correct)** *Let $a, b : 2$ be two bits. Then*

$$\lceil [\![\mathsf{half\text{-}adder}]\!] \langle a, b\rangle \rceil_{2^2} = \lceil a \rceil_2 + \lceil b \rceil_2$$

We have proven this theorem in Coq. This particular theorem can be proven by exhaustive case analysis (there are only four cases) and equational reasoning using Simplicity's denotational semantics.

We continue and define a full adder by combining two half adders. From there we can build ripple-carry adders for larger word sizes by combining full

adders from smaller word sizes. The interested reader can find the Simplicity expressions for these full adders in Appendix B.

Continuing, we can build multipliers and other bit-manipulation functions. These can be combined to implement more sophisticated functions.

We have written the SHA-256 block compression function in Simplicity. Using 256-bit arithmetic, we have also constructed expressions for elliptic curve operations over the Secp256k1 curve [9] that Bitcoin uses, and we have built a form of ECDSA signature validation [23] in Simplicity.

To gain an understanding of the size of Simplicity expressions, let us examine our implementation of the SHA-256 block compression function, $\mathsf{sha\text{-}256\text{-}block} : 2^{256} \times 2^{512} \vdash 2^{256}$. Our Simplicity expression consists of 3 274 442 combinators. However, this counts the total number of nodes in the abstract syntax tree that makes up the expression. Several sub-expressions of this expression are duplicated and can be shared. Imagine taking the abstract syntax tree and sharing identical sub-expressions to create a directed acyclic graph (DAG) representing the same expression. Counting this way, the same expression contains only 1 130 unique typed sub-expressions, which is the number of nodes that would occur in the DAG representing the expression.

Taking advantage of shared sub-expressions is critical because it makes the representation of typical expressions exponentially smaller in size. We choose to leave sub-expression sharing implicit in Simplicity's term language. This ensures that Simplicity's semantics are not affected by how sub-expressions are shared. However, we do transmit and store Simplicity expressions in a DAG format. This DAG representation of expressions is also important when we consider static analysis in Section 3.3.

Using our formal semantics of Simplicity in Coq, we have proven our implementation of SHA-256 is correct.

**Theorem 2.3 (SHA-256 Correct)** *Let* $a : 2^{256}$ *and* $b : 2^{512}$. *Let* $\mathsf{sha\text{-}256\text{-}block} : 2^{256} \times 2^{512} \vdash 2^{256}$ *be our Simplicity implementation of the SHA-256 block compression function. Let* $\mathsf{SHA256Block} : 2^{256} \times 2^{512} \to 2^{256}$ *be the SHA-256 block compression function. Then*

$$[\![\mathsf{sha\text{-}256\text{-}block}]\!]\langle a, b\rangle = \mathsf{SHA256Block}\langle a, b\rangle$$

Our reference implementation of the SHA-256 compression function in Coq[1] is taken from the Verified Software Toolchain (VST) project [3] where they use it as their reference implementation for proving OpenSSL's C implementation of SHA-256 correct [2]. If we combine our Coq proof with the VST project's Coq proof, we would get a formal proof that our Simplicity implementation of SHA-256 matches OpenSSL's C implementation of SHA-256.

[1]The type of the reference implementation for the SHA-256 compression function in Coq actually uses lists of 32-bit words. To simplify the presentation here, we have omitted the translation between representations of the compression function's inputs and outputs.

# 3 The Bit Machine

One of the design goals for Simplicity is to be able to compute a bound on the computation costs of evaluating a Simplicity program. While the denotational semantics of Simplicity are adequate for determining the functional behavior of Simplicity programs, we need operational semantics to provide a measure of computational resources used. To do this, we create an abstract machine, called the Bit Machine, tailored for evaluating Simplicity programs.

The Bit Machine is an abstract imperative machine whose state consists of two non-empty stacks of data frames. One stack holds read-only frames, and the other stack holds write-only frames. A data frame consists of an array of cells and a cursor pointing either at one of the cells or pointing just beyond the end of the array. Each cell contains one of three values: a zero bit, a one bit, or an undefined value. The Bit Machine has a set of instructions that manipulate the two stacks and their data frames.

Notionally, we will write a frame like in the following example.

$$[\mathtt{01}\underline{\mathtt{0}}\mathtt{?1?1}]$$

This frame contains 7 cells. The ? character denotes a cell with an undefined value. The `0` and `1` characters denotes cells with the corresponding bit value. The frame's cursor is pointing to the underscored cell.

When a frame's cursor is pointing past the end of the array, we will write an underscore after the array's cells like the following.

$$[\mathtt{010?1?1}\underline{\ }]$$

Table 1 illustrates an example of a possible state of the Bit Machine. This example contains a read-frame stack that has four data frames and a write-frame stack that has two data frames. The top frame of the read-frame stack is called the *active read frame*, and the top frame of the write-frame stack is called the *active write frame*.

| read frame stack | write frame stack |
|---|---|
| $[\mathtt{100}\underline{\mathtt{1}}\mathtt{??110101000}]$ | $[\mathtt{11??1101}\underline{\ }]$ |
| $[\underline{\mathtt{0}}\mathtt{000}]$ | $[\mathtt{111}\underline{\mathtt{?}}\mathtt{?}]$ |
| $[\underline{\ }]$ | |
| $[\underline{\mathtt{1}}\mathtt{0}]$ | |

Table 1: Example state for the Bit Machine

The Bit Machine has ten instructions to manipulate its state, which we describe below.

- **nop**: This instruction doesn't change the state of the machine.

- **write**($b$): This instruction writes bit value $b$ to the cell under the active write frame's cursor and advances the cursor by one cell.
  - The value $b$ must be either `0` or `1` or else the machine crashes.
  - The active write frame's cursor must not be beyond the end of the array or else the machine crashes.
  - The cell written to must have an undefined value before writing to it or else the machine crashes.
- **copy**($n$): This instruction copies $n$ cells from the active read frame, starting at its cursor, to the active write frame, starting at its cursor, and advances the active write frame's cursor by $n$ cells.
  - There must be at least $n$ cells between the active read frame's cursor and the end of its array or else the machine crashes.
  - There must be at least $n$ cells between the active write frame's cursor and the end of its array and they all must have an undefined value before writing to them or else the machine crashes.
  - Note that undefined values can be copied.
- **skip**($n$): This instruction advances the active write frame's cursor by $n$ cells.
  - This instruction is allowed to advance the cursor to the point just past the end of the array; however if $n$ is large enough that it would advance it past this point, the machine crashes instead.
- **fwd**($n$): This instruction moves the active read frame's cursor forward by $n$ cells.
  - This instruction is allowed to advance the cursor to the point just past the end of the array; however if $n$ is large enough that it would advance it past this point, the machine crashes instead.
- **bwd**($n$): This instruction moves the active read frame's cursor backwards by $n$ cells.
  - If this instruction would move the cursor before the beginning of the array, the machine crashes instead.
- **newFrame**($n$): This instruction allocates a new frame with $n$ cells and pushes it onto the write-frame stack, making it the new active write frame.
  - The cursor for this new frame starts at the beginning of the array.
  - The cells of the new frame are initialized with undefined values.
- **moveFrame**: This instruction pops a frame off the write-frame stack and pushes it onto the read-frame stack.

  - The moved frame has its cursor reinitialized to the beginning of the array.
  - If this would leave the machine with an empty write frame stack, the machine crashes instead.

- **dropFrame**: This instruction pops a frame off the read-frame stack and deallocates it.
  - If this would leave the machine with an empty read-frame stack, the machine crashes instead.

- **read**: This instruction doesn't modify the state of the machine. Instead it returns the value of the bit under the active read frame's cursor to the machine's operator.
  - if the value under the active read frame's cursor is undefined, or if it is past the end of its array, the machine crashes instead.

The Bit Machine is deliberately designed to crash at anything resembling undefined behavior.

## 3.1 Bit Machine Values

We can use the Bit Machine to evaluate Simplicity programs on their inputs. First, we define how many cells are needed to hold the values of a given type.

$$\begin{aligned} \mathsf{bitSize}(\mathbb{1}) &\coloneqq 0 \\ \mathsf{bitSize}(A + B) &\coloneqq 1 + \mathsf{max}(\mathsf{bitSize}(A), \mathsf{bitSize}(B)) \\ \mathsf{bitSize}(A \times B) &\coloneqq \mathsf{bitSize}(A) + \mathsf{bitSize}(B) \end{aligned}$$

The unit type doesn't need any bits to represent its one value. A value of sum type needs one bit for its tag and then enough bits to represent either of its left or right values. A value of product type needs space to hold both its values.

We precisely define a representation of values for Simplicity types as array of cells below:

$$\begin{aligned} \ulcorner \langle\rangle \urcorner_{\mathbb{1}} &\coloneqq [] \\ \ulcorner \sigma^{\mathbf{L}}(a) \urcorner_{A+B} &\coloneqq [\mathsf{0}] \cdot [?]^{\mathsf{padl}(A,B)} \cdot \ulcorner a \urcorner_A \\ \ulcorner \sigma^{\mathbf{R}}(b) \urcorner_{A+B} &\coloneqq [\mathsf{1}] \cdot [?]^{\mathsf{padr}(A,B)} \cdot \ulcorner b \urcorner_B \\ \ulcorner \langle a, b \rangle \urcorner_{A \times B} &\coloneqq \ulcorner a \urcorner_A \cdot \ulcorner b \urcorner_B \end{aligned}$$

For the notation above, $\cdot$ denotes the concatenation of arrays of cells and $[b]^n$ denotes an array of $n$ cells all containing $b$.

The padding for left and right values fills the array with sufficient undefined values so that the array ends up with the required length.

$$\mathsf{padl}(A, B) \coloneqq \mathsf{max}(\mathsf{bitSize}(A), \mathsf{bitSize}(B)) - \mathsf{bitSize}(A)$$
$$\mathsf{padr}(A, B) \coloneqq \mathsf{max}(\mathsf{bitSize}(A), \mathsf{bitSize}(B)) - \mathsf{bitSize}(B)$$

Below are some examples of values represented as arrays of cells for the Bit Machine. We define the inverse of $\lceil\cdot\rceil_{2^n}$ to be $\lfloor\cdot\rfloor_{2^n}$ that maps numbers to the values that represent them.

$$\ulcorner\sigma^{\mathbf{L}}(\lfloor 3\rfloor_{2^2})\urcorner_{2^2+2} = [011]$$
$$\ulcorner\sigma^{\mathbf{R}}(\lfloor 0\rfloor_{2})\urcorner_{2^2+2} = [1?0]$$

The array of cells contains a list of the tags from the sum types and that is about it. Notice that values of a multi-bit word type, $2^n$, have exactly $n$ cells and the representation is identical to its big-endian binary representation.

## 3.2 Operational Semantics

The operational semantics for Simplicity are given by recursively translating a Simplicity expression into a sequence of instructions for the Bit Machine. Figure 2 defines $\langle\!\langle t : A \vdash B \rangle\!\rangle^*$, which results in a sequence of instructions for the Bit Machine that evaluates the function $[\![t]\!]$.

We have formalized the Bit Machine in Coq and Figure 2's translation of Simplicity expressions into Bit Machine instructions. We have verified, with Coq, that our Bit Machine implementation of Simplicity respects Simplicity's denotational semantics.

**Theorem 3.1 (Correctness of Operational Semantics)** *Let $t : A \vdash B$ be any Simplicity expression. Let $v : A$ be any value of type $A$. Initialize a Bit Machine with*

- *the value $\ulcorner v \urcorner_A$ as the only frame on the read-frame stack with its cursor at the beginning of the frame, and*
- *the value $[?]^{\mathsf{bitSize}(B)}$ as the only frame on the write-frame stack with its cursor at the beginning of the frame.*

*After executing the instructions $\langle\!\langle t : A \vdash B \rangle\!\rangle^*$, the final state of the Bit Machine has*

- *the value $\ulcorner v \urcorner_A$ as the only frame on the read-frame stack, with its cursor at the beginning of the frame, and*
- *the value $\ulcorner [\![t]\!](v) \urcorner_B$ as the only frame on the write-frame stack, with its cursor at the end of the frame.*

$$
\begin{aligned}
\langle\!\langle \mathsf{iden} : A \vdash A \rangle\!\rangle^* &\coloneqq \mathbf{copy}(\mathsf{bitSize}(A)) \\
\langle\!\langle \mathsf{comp}\, s\, t : A \vdash C \rangle\!\rangle^* &\coloneqq \mathbf{newFrame}(\mathsf{bitSize}(B)); \langle\!\langle s : A \vdash B \rangle\!\rangle^*; \\
&\quad\ \mathbf{moveFrame}; \langle\!\langle t : B \vdash C \rangle\!\rangle^*; \\
&\quad\ \mathbf{dropFrame} \\
\langle\!\langle \mathsf{unit} : A \vdash \mathbb{1} \rangle\!\rangle^* &\coloneqq \mathbf{nop} \\
\langle\!\langle \mathsf{injl}\, t : A \vdash B + C \rangle\!\rangle^* &\coloneqq \mathbf{write}(\mathsf{0}); \mathbf{skip}(\mathsf{padl}(B, C)); \langle\!\langle t : A \vdash B \rangle\!\rangle^* \\
\langle\!\langle \mathsf{injr}\, t : A \vdash B + C \rangle\!\rangle^* &\coloneqq \mathbf{write}(\mathsf{1}); \mathbf{skip}(\mathsf{padr}(B, C)); \langle\!\langle t : A \vdash C \rangle\!\rangle^* \\
\langle\!\langle \mathsf{case}\, s\, t : (A + B) \times C \vdash D \rangle\!\rangle^* &\coloneqq \text{match } \mathbf{read} \text{ with} \\
&\quad \begin{cases}
\mathsf{0} \to \mathbf{fwd}(1 + \mathsf{padl}(A, B)); \\
\qquad \langle\!\langle s : A \times C \vdash D \rangle\!\rangle^*; \\
\qquad \mathbf{bwd}(1 + \mathsf{padl}(A, B)) \\
\mathsf{1} \to \mathbf{fwd}(1 + \mathsf{padr}(A, B)); \\
\qquad \langle\!\langle t : B \times C \vdash D \rangle\!\rangle^*; \\
\qquad \mathbf{bwd}(1 + \mathsf{padr}(A, B))
\end{cases} \\
\langle\!\langle \mathsf{pair}\, s\, t : A \vdash B \times C \rangle\!\rangle^* &\coloneqq \langle\!\langle s : A \vdash B \rangle\!\rangle^*; \langle\!\langle t : A \vdash C \rangle\!\rangle^* \\
\langle\!\langle \mathsf{take}\, t : A \times B \vdash C \rangle\!\rangle^* &\coloneqq \langle\!\langle t : A \vdash C \rangle\!\rangle^* \\
\langle\!\langle \mathsf{drop}\, t : A \times B \vdash C \rangle\!\rangle^* &\coloneqq \mathbf{fwd}(\mathsf{bitSize}(A)); \langle\!\langle t : B \vdash C \rangle\!\rangle^*; \\
&\quad\ \mathbf{bwd}(\mathsf{bitSize}(A))
\end{aligned}
$$

Figure 2: Operational Semantics for Simplicity using the Bit Machine.

*In particular, the Bit Machine never crashes during this execution.*

The fact that the Bit Machine never crashes means that during execution it never reads from an undefined cell, nor does it ever write to a defined cell. This means that, if one implements the Bit Machine on a real computer, one can use an ordinary array of bits to represent the array of cells and cells that are supposed to hold undefined values can be safely set to any bit value. As long as this implementation only executes instructions translated from Simplicity and started from a proper initial state, the resulting computation will be the same as if one had used an explicit representation for undefined values.

### 3.3 Static Analysis

Using the Bit Machine, we can measure computational resources needed by a Simplicity program. For instance we can:

- count the number of instructions executed by the Bit Machine.
- count the number of cells copied by the Bit Machine.

$$
\begin{aligned}
\mathsf{extraCellBnd}(\mathsf{iden} : A \vdash A) &\coloneqq 0 \\
\mathsf{extraCellBnd}(\mathsf{comp}\, s\, t : A \vdash C) &\coloneqq \mathsf{bitSize}(B) + \\
&\qquad \mathsf{max}(\mathsf{extraCellBnd}(s : A \vdash B), \\
&\qquad\quad \mathsf{extraCellBnd}(t : B \vdash C)) \\
\mathsf{extraCellBnd}(\mathsf{unit} : A \vdash \mathbb{1}) &\coloneqq 0 \\
\mathsf{extraCellBnd}(\mathsf{injl}\, t : A \vdash B + C) &\coloneqq \mathsf{extraCellBnd}(t : A \vdash B) \\
\mathsf{extraCellBnd}(\mathsf{injr}\, t : A \vdash B + C) &\coloneqq \mathsf{extraCellBnd}(t : A \vdash C) \\
\mathsf{extraCellBnd}(\mathsf{case}\, s\, t : (A + B) \times C \vdash D) &\coloneqq \mathsf{max}(\mathsf{extraCellBnd}(s : A \times C \vdash D), \\
&\qquad \mathsf{extraCellBnd}(t : B \times C \vdash D)) \\
\mathsf{extraCellBnd}(\mathsf{pair}\, s\, t : A \vdash B \times C) &\coloneqq \mathsf{max}(\mathsf{extraCellBnd}(s : A \vdash B), \\
&\qquad \mathsf{extraCellBnd}(t : A \vdash C)) \\
\mathsf{extraCellBnd}(\mathsf{take}\, t : A \times B \vdash C) &\coloneqq \mathsf{extraCellBnd}(t : A \vdash C) \\
\mathsf{extraCellBnd}(\mathsf{drop}\, t : A \times B \vdash C) &\coloneqq \mathsf{extraCellBnd}(t : B \vdash C)
\end{aligned}
$$

$$
\mathsf{cellBnd}(t : A \vdash B) \coloneqq \mathsf{bitSize}(A) + \mathsf{bitSize}(B) + \mathsf{extraCellBnd}(t : A \vdash B)
$$

Figure 3: Definition of cellBnd.

- count the maximum number of cells in both stacks at any point during execution.
- count the maximum number of frames in both stacks at any point during execution.

The first two are measurements of time used by the Bit Machine, and the last two are measurements of space used by the Bit Machine.

Using simple static analysis, we can quickly compute an upper bound on these sorts of resource costs. In this paper, we will focus on the example of counting the maximum number of cells in both stacks at any point during execution. Figure 3 defines a function cellBnd that computes an upper bound on the maximum number of cells that are needed during execution.

**Theorem 3.2 (Static Analysis of Cell Usage)** *Let $t : A \vdash B$ be any Simplicity expression. Let $v : A$ be any value of type $A$. Initialize a Bit Machine as specified in Theorem 3.1. The maximum number of cells in both stacks of the Bit Machine at any point during the execution of $\langle\!\langle t : A \vdash B \rangle\!\rangle^*$ from this initial state never exceeds* $\mathsf{cellBnd}(t : A \vdash B)$.

We have formalized the above static analysis and proven the above theorem in Coq.

These kinds of static analyses are simple recursive functions of Simplicity expressions, and the intermediate results for sub-expressions can be shared. By using a DAG for the in-memory representation of Simplicity expressions, we can transparently cache these intermediate results. This means the time needed to compute static analysis is proportional to the size of the DAG representing the Simplicity expression, as opposed to the time needed for dynamic analysis such as evaluation, which may take time proportional to the size of the tree representing the Simplicity expression.

The Bit Machine is an abstract machine, so we can think of these sorts of static analyses as bounding abstract resource costs. As long as the abstract resource costs are limited, then the resource costs of any implementation of Simplicity will also be limited. These sorts of precise static analyses of resource costs are more sophisticated than what is currently available for Bitcoin Script.

Notice that the definition of $\mathsf{cellBnd}$ does not directly reference the Bit Machine, so limits on the bounds computed by these static analyses can be imposed on protocols that use Simplicity without necessarily requiring that applications use the Bit Machine for evaluation. While we do use an implementation of the Bit Machine in our prototype Simplicity evaluator written in C, others are free to explore other models of evaluation. For example, the Bit Machine copies data to implement the $\mathsf{iden}$ combinator, but another model of evaluation might create shared references to data instead.

### 3.3.1 Tail Composition Optimization

Tail call optimization is an optimization used in many languages where a procedure's stack frame is freed prior to a call to another procedure when that call is the last command of the procedure. The $\mathsf{comp}$ combinator in Simplicity behaves much like a procedure call, and we can implement an analogous optimization for the translation of Simplicity to the Bit Machine. Interested readers can find this tail composition optimized translation, $\langle\!\langle t : A \vdash B \rangle\!\rangle^{\mathsf{tco}}_{\mathsf{off}}$, defined in Appendix C, along with a static analysis of its memory use.

## 3.4 Jets

Evaluation of a Simplicity expression with the Bit Machine recursively traverses the expression. Before evaluating some sub-expression $t : A \vdash B$, the Bit Machine is always in a state where the active read frame is of the form

$$r_1 \cdot \ulcorner v \urcorner_A \cdot r_2$$

for some value $v : A$ and some arrays $r_1$ and $r_2$, and where the cursor is placed at the beginning of the $\ulcorner v \urcorner_A$ array slice.

Furthermore the active write frame is of the form

$$w_1 \cdot [?]^{\mathsf{bitSize}(B)} \cdot w_2$$

for some arrays $w_1$ and $w_2$, and where the cursor is placed at the beginning of the $[?]^{\mathsf{bitSize}(B)}$ array slice.

After the evaluation of the sub-expression $t : A \vdash B$, the active write frame is of the form

$$w_1 \cdot \ulcorner [\![t]\!](v) \urcorner_B \cdot w_2$$

and where the cursor is placed at the beginning of the $w_2$ array slice.

Taking an idea found in Urbit [32], we notice that if we recognize a familiar sub-expression $t : A \vdash B$, the interpreter may bypass the BitMachine's execution of $\langle\!\langle t : A \vdash B \rangle\!\rangle^*$ and instead directly compute and write $\ulcorner [\![t]\!](v) \urcorner_B$ to the active write frame. Following Urbit, we call such a familiar expression and the code that replaces it a *jet*.

Jets are essential for making Simplicity a practical language. For our Blockchain application we expect to have jets for at least the following expressions:

- arithmetic operations (addition, subtraction, and multiplication) from 8-bit to 256-bit word size,
- comparison operations (less than, less than or equal to, equality) from 8-bit to 256-bit word size,
- logical bitwise operation for 8-bit to 256-bit word sizes,
- constant functions for every possible 8-bit word,
- compression functions from hash functions such as SHA-256's compression function,
- elliptic curve point operations for the Secp256k1 curve [9], and
- digital signature validation for ECDSA [23] or Schnorr [28] signatures.

We take advantage of the fact that the representation of the values used for arithmetic expressions in the Bit Machine match the binary memory format for real hardware. This lets us write these jets by directly reading from and writing to data frames.

Jets have several nice properties:

- Jets provide a formal specification of their behavior. The implementation of a jet must produce output identical to the output that would be produced by the Simplicity expression being replaced. There is no possibility for an ambiguous interpretation of what a jet computes.
- Jets cannot accidentally introduce new behavior or new side effects because they can only replicate the behavior of Simplicity expressions. To add new behavior to Simplicity we must explicitly extend Simplicity (see Section 4).
- Jets are transparent when it comes to reasoning about Simplicity expressions. Jets are logically equal to the code they replace. Therefore, when proving properties of one's Simplicity code, jets can safely be ignored.

Naturally, we expect jetted expressions to have properties already proven and available; this will aid reasoning about Simplicity programs that make use of jets.

Because jets are transparent, the static analyses of resource costs are not affected by their existence. To encourage the use of jets, we anticipate discounts to be applied to the resource costs of programs that use jets based on the estimated savings of using jets.

When a suitably rich set of jets is available, we expect the bulk of the computation specified by a Simplicity program to be made up of these jets, with only a few combinators used to combine the various jets. This should bring the computational requirements needed for Simplicity programs in line with existing blockchain languages. In light of this, one could consider Simplicity to be a family of languages, where each language is defined by a set of jets that provide computational elements tailored for its particular application.

# 4 Integration with Blockchains

Core Simplicity is language that does pure computation. In order to interact with blockchains we extend Simplicity with new combinators.

## 4.1 Transaction Digests

The main operation used in Bitcoin Script is OP_CHECKSIG which, given a public key, a digital signature, and a SigHash type [15], generates a digest of the transaction in accordance with the SigHash type and validates that the digital signature for the digest is correct for the given public key. This operation allows users to create programs that require their signatures for their transactions to be authorized.

In core Simplicity we can implement, and jet, the digital signature validation algorithm. However, it is not possible to generate the transaction digest because core Simplicity doesn't have access to the transaction data. To remedy this, we add a new primitive combinator to Simplicity

$$\mathsf{sighash} : \mathsf{SigHashType} \vdash 2^{256}$$

where $\mathsf{SigHashType} \coloneqq (\mathbb{1} + \mathbb{2}) \times \mathbb{2}$ is a Simplicity type suitable for encoding all possible SigHash types.

This combinator returns the digest of the transaction for the given SigHash type. Together with the Simplicity implementation of digital signature verification, we can implement the equivalent of Bitcoin's OP_CHECKSIG.

We can add other primitives to Simplicity to access specific fields of transaction data in order to implement features such as Bitcoin's timelock operations [7, 30]. While Simplicity avoids providing direct access to persistent storage, one could imagine an application where transactions contain data for transactional updates to a persistent store. Simplicity could support primitives

to read the details of these transactional updates and thereby enforce any set of programmable rules on them.

## 4.2 Merklized Abstract Syntax Tree

Recall that when using P2SH, users commit to their program by hashing it and placing that hash in the outputs of transactions. Only when redeeming their funds does the user reveal their program, whose hash must match the committed hash.

*Pay to Merklized Abstract Syntax Tree*, or *P2MAST* [17], enhances the P2SH idea. Instead of hashing a linear encoding of a Simplicity expression, we use the SHA-256's block compression function, $\mathsf{SHA256Block} : 2^{256} \times 2^{512} \to 2^{256}$, to recursively hash Simplicity sub-expressions. This computes a Merkle root of Simplicity's abstract syntax tree that we denote by $\#(\cdot)$ and define as

$$\begin{aligned}
\#(\mathsf{iden}) &\coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{iden}}, \lfloor 0 \rfloor_{2^{512}} \rangle \\
\#(\mathsf{comp}\, s\, t) &\coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{comp}}, \langle \#(s), \#(t) \rangle \rangle \\
\#(\mathsf{unit}) &\coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{unit}}, \lfloor 0 \rfloor_{2^{512}} \rangle \\
\#(\mathsf{injl}\, t) &\coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{injl}}, \langle \#(t), \lfloor 0 \rfloor_{2^{256}} \rangle \rangle \\
\#(\mathsf{injr}\, t) &\coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{injr}}, \langle \#(t), \lfloor 0 \rfloor_{2^{256}} \rangle \rangle \\
\#(\mathsf{case}\, s\, t) &\coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{case}}, \langle \#(s), \#(t) \rangle \rangle \\
\#(\mathsf{pair}\, s\, t) &\coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{pair}}, \langle \#(s), \#(t) \rangle \rangle \\
\#(\mathsf{take}\, t) &\coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{take}}, \langle \#(t), \lfloor 0 \rfloor_{2^{256}} \rangle \rangle \\
\#(\mathsf{drop}\, t) &\coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{drop}}, \langle \#(t), \lfloor 0 \rfloor_{2^{256}} \rangle \rangle \\
\#(\mathsf{sighash}) &\coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{sighash}}, \lfloor 0 \rfloor_{2^{512}} \rangle
\end{aligned}$$

where $\mathsf{tag}_c : 2^{256}$ is an appropriate unique set of initial vectors per combinator. We use the SHA-256 hash of the combinator name for its $\mathsf{tag}$ value.

When computing Merkle roots, like other kinds of static analysis, the intermediate results of sub-expressions can be shared. Bitcoin Script cannot share sub-expressions in this manner due to the linear encoding of Bitcoin Script programs.

Another advantage of Merkle roots is that unused branches of $\mathsf{case}$ expressions can be pruned at redemption time. Each unused branch can be replaced with the value of its Merkle root. This saves on bandwidth, storage costs, and enhances privacy by not revealing more of a Simplicity program than necessary. Again, this is something not possible in Bitcoin Script.

To enable redemption of pruned Simplicity expressions, we add two new combinators to replace the $\mathsf{case}$ combinator when pruning.

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

These assertion combinators follow the same form as the $\mathsf{case}$ combinator, except one branch is replaced by a hash. The semantics of these combinators require that the first component of their input be $\sigma^{\mathbf{L}}$ or $\sigma^{\mathbf{R}}$ as appropriate and then they evaluate the available branch. The $h : 2^{256}$ values do not affect the semantics; they instead influence the Merkle root computation.

$$\#(\mathsf{assertl}\, s\, h) \coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{case}}, \langle \#(s), h\rangle\rangle$$
$$\#(\mathsf{assertr}\, h\, t) \coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{case}}, \langle h, \#(t)\rangle\rangle$$

Notice that we use $\mathsf{tag}_{\mathsf{case}}$ for the tags of the assertions. During redemption, one can replace $\mathsf{case}$ statements with appropriate assertions while still matching the committed Merkle root of the whole expression.

$$\#(\mathsf{case}\, s\, t) = \#(\mathsf{assertl}\, s\, \#(t)) = \#(\mathsf{assertr}\, \#(s)\, t)$$

As long as the assertions hold, the computation remains unchanged. If an assertion fails, because you pruned off a branch that was actually necessary, then the computation fails and a transaction trying to redeem funds in that manner is invalid.

Because failure is now a possible result during evaluation, we add a combinator $\mathsf{fail}$ to allow one to develop programs that use assertions.

$$\mathsf{fail} : A \vdash B$$

$$\#(\mathsf{fail}) \coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{fail}}, \lfloor 0 \rfloor_{2^{512}}\rangle$$

When combined with the signature verification expression, one can build the equivalent of Bitcoin's $\mathsf{OP\_CHECKSIGVERIFY}$ operation to assert that a signature is valid.

Degenerate assertions can also be used to enhance privacy by mixing entropy into Merkle roots of sub-expressions. Given $t : A \vdash B$ the expression

$$\mathsf{comp}\,(\mathsf{pair}\,(\mathsf{injl}\,\mathsf{iden})\ \mathsf{unit})\,(\mathsf{assertl}\,(\mathsf{take}\, t)\, h) : A \vdash B$$

is semantically identical to $t$, but mixes $h$ into its Merkle root computation. When used inside branches that are likely to be pruned, this prevents adversaries from effectively grinding out Simplicity expressions to see if they match the hash of the missing branch.

The Merkle root of an expression does not commit to its types. We use first-order unification [27] to perform type inference on Simplicity expressions, replacing any remaining type variables with the unit type. Because the types of pruned branches are discarded, the inferred types may end up smaller than in the originally committed program. When this happens, the memory requirements for evaluation with the Bit Machine may also decrease.

## 4.3 Witness Values

During redemption, the user must provide inputs, such as digital signatures and other witness data, in order to authorize a transaction. Rather than passing this input as an argument to the Simplicity program, we embed such witness values directly into the Simplicity expressions using the witness combinator.

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

Semantically, the witness combinator just denotes a constant function, which is something we can already write in core Simplicity. The difference lies in its Merkle root computation.

$$\#(\mathsf{witness}\, b) \coloneqq \mathsf{SHA256Block}\langle \mathsf{tag}_{\mathsf{witness}}, \lfloor 0 \rfloor_{2^{512}} \rangle$$

The value $b$ is not committed by the Merkle root. This allows the user to set witness value at redemption time, without affecting the expression's Merkle root. Users can use witness combinators at places where digital signatures are needed and at places where choices are made at redemption time.

Using witness combinators enhances privacy because they are pruned away when they occur in unused branches. This leaves little evidence that there was an optional input at all.

The amount of witness data allowed at redemption time is limited at commitment time because each witness data type is finite and there can only be a finite number of witness combinators committed. This limits the computational power of Simplicity. For example, it is not possible to create a Simplicity program that allows hashing of an unbounded amount of witness data in order to validate a digital signature. However, it is possible to create a Simplicity program that allows a large, but bounded, amount of witness data to be hashed that is more than will ever be needed in practice. Thanks to pruning, only the code used to hash the amount of witness data that actually occurs need be provided at redemption time.

Type inference determines the minimal type for witness values. This prevents them from being filled with unnecessary data.

The witness combinator is not allowed to be used in jets.

## 4.4 Extended Simplicity Semantics

In order to accommodate these extensions to core Simplicity, we need to broaden Simplicity's semantics. We use a monad, $\mathfrak{M}$, to capture the new effects of these extensions.

$$\mathfrak{M}(A) \coloneqq \mathsf{Trans} \to A_{\perp}$$

This monad is a combination of an environment monad (also called a reader monad) with an exception monad.

This monad provides implicit access to a value of $\mathsf{Trans}$ type, which we define to be the type of transaction data for the particular blockchain. During evaluation, it is filled with the transaction containing the input redeeming the Simplicity program being evaluated. This is used to give semantics to the $\mathsf{sighash}$ and other similar primitives.

The monad also adds a failure value to the result type, denoted by $\bot : A_\bot$ This allows us to give semantics to assertions and $\mathsf{fail}$.

Given an expression $t : A \vdash B$, we denote this extended semantics by $[\![t]\!]_{\mathfrak{M}} : A \to \mathfrak{M}(B)$. The core Simplicity semantics are lifted in the natural way so that

$$[\![t]\!]_{\mathfrak{M}}(a) = \lambda e : \mathsf{Trans}.[\![t]\!](a)$$

holds whenever $t$ is composed of only core Simplicity combinators. The extended set of combinators have the following semantics

$$\begin{aligned}
[\![\mathsf{sighash}]\!]_{\mathfrak{M}}(a) &\coloneqq \lambda e.\, \mathsf{MakeSigHash}(a, e)\\
[\![\mathsf{assertl}\ s\ h]\!]_{\mathfrak{M}}\langle \sigma^{\mathbf{L}}(a), c\rangle &\coloneqq \lambda e.[\![s]\!]_{\mathfrak{M}}\langle a, c\rangle(e)\\
[\![\mathsf{assertl}\ s\ h]\!]_{\mathfrak{M}}\langle \sigma^{\mathbf{R}}(b), c\rangle &\coloneqq \lambda e.\bot\\
[\![\mathsf{assertr}\ h\ t]\!]_{\mathfrak{M}}\langle \sigma^{\mathbf{L}}(a), c\rangle &\coloneqq \lambda e.\bot\\
[\![\mathsf{assertr}\ h\ t]\!]_{\mathfrak{M}}\langle \sigma^{\mathbf{R}}(b), c\rangle &\coloneqq \lambda e.[\![t]\!]_{\mathfrak{M}}\langle b, c\rangle(e)\\
[\![\mathsf{fail}]\!]_{\mathfrak{M}}(a) &\coloneqq \lambda e.\bot\\
[\![\mathsf{witness}\ b]\!]_{\mathfrak{M}}(a) &\coloneqq \lambda e.b
\end{aligned}$$

where $\mathsf{MakeSigHash}(a, e)$ is a function that returns a hash of a given transaction $e$ in accordance with the given SigHash type $a$.

The operational semantics are also extended to support our new combinators by providing the Bit Machine with access to the transaction data and by adding an explicit **crash** instruction to support assertions and $\mathsf{fail}$.

We note that the effects captured by our monad are commutative and idempotent.[2] While Simplicity's operational semantics implicitly specify an order of evaluation, the extended denotational semantics are independent of evaluation order. This simplifies formal reasoning about Simplicity programs that use the extended semantics. That said, we expect the majority of Simplicity program's sub-expressions to be written in core Simplicity whose denotational semantics contain no effects at all, making reasoning about them simpler still.

## 4.5 Using Simplicity Programs in Blockchains

Simplicity programs are Simplicity expressions of type $p : \mathbb{1} \vdash \mathbb{1}$, that may use any of our extended combinators. Users transact by constructing their Simplic-

[2] We use the terms 'commutative' and 'idempotent' in the sense of King and Wadler [16] as opposed to the traditional category theoretic definition of an idempotent monad.

ity program and computing its Merkle root $\#(p)$. They have their counterparty create a transaction that sends funds to that hash.

Later, when a user wants to redeem their received funds, they create a transaction whose input contains a Simplicity program whose Merkle root matches the previous hash. At this point they have the ability to set the witness values and prune any unneeded branches from case expressions. The witness combinators handle the program's effective inputs, so the program $p$ is evaluated at the value $\langle\rangle : \mathbb{1}$. No output is needed because the program can use assertions to fail in case of invalid witnesses. If the execution completes successfully, without failing, the transaction is considered valid.

The basic signature program that mimics Bitcoin's basic signature program is composed of the following core Simplicity expressions

$$\mathsf{checkSig} : \mathsf{Signature} \times (\mathsf{PubKey} \times 2^{256}) \vdash 2$$
$$\mathsf{pubKey} : A \vdash \mathsf{PubKey}$$

where checkSig checks whether a given signature validates for a given public key and message hash, and pubKey returns a user's specific public key.

These expressions can be combined into a basic signature verification program.

$$\begin{aligned}\mathsf{basicSigVerify}\, b\, c \coloneqq \mathsf{comp}\, (\mathsf{pair}\, (\mathsf{witness}\, b)\\ (\mathsf{pair}\, \mathsf{pubKey}\, (\mathsf{comp}\, (\mathsf{witness}\, c)\, \mathsf{sighash})))\\ (\mathsf{comp}\, (\mathsf{pair}\, \mathsf{checkSig}\, \mathsf{unit})\, (\mathsf{case}\, \mathsf{fail}\, \mathsf{unit}))\end{aligned}$$

Other, more complex programs can be built to perform multi-signature checks, to implement sophisticated smart contracts such as zero-knowledge contingent payments, or to create novel smart contracts.

# 5 Results and Future Work

In many ways, Simplicity is best characterized by what features it leaves out rather than what features it contains.

- Simplicity has no state. Purely functional, expression-based languages facilitate equational reasoning about the semantics of expressions. For example, there are no concerns about aliased references to a global heap, so there is no need to work with separation logic or Hoare logic.
- Simplicity has no named variables. Using combinators lets us avoid dealing with binders and environments for bound variables. This helps keep our interpreter and static analyses simple and further eases equational reasoning about Simplicity expressions.

- Simplicity has no function types and therefore no higher-order functions. While it is possible to compute upper bounds of computation resources of expressions in the presence of function types, it likely that those bounds would be so far above their actual needs that such analysis would not be useful.

- Simplicity has no unbounded loops or recursion. It is possible to build smart contracts with state carried through loops using covenants [25], without requiring unbounded loops within Simplicity itself. Bounded loops, such as the 64 rounds needed by our SHA-256 implementation, can be achieved by unrolling the loop. Because of sub-expression sharing, this doesn't unreasonably impact program size. We do not directly write Simplicity, rather we use functions written in Coq or Haskell to generate Simplicity. These languages do support recursion and we use loops in these meta-languages to generate unrolled loops in Simplicity.

Throughout this paper we have noted which theorems we have verified with Coq. Other proofs are under development. In particular, we plan to formally verify the denotational and operational semantics the full Simplicity language, including assertions and blockchain primitives.

To validate the suitability of Simplicity, we will be building example smart contracts in a test blockchain or sidechain [4] application using Simplicity.

We also plan to use the VST project [3] to prove the correctness of our C implementation of the Bit Machine. This would let us formally verify that the assembly code generated by the CompCert compiler [18] from our C implementation respects Simplicity's formal semantics. In particular, we would be able to prove that substituting jets, such as SHA-256, with fast C or assembly implementations preserves the semantics of Simplicity.

Simplicity is designed as a low-level language interpreted by blockchain software. We anticipate higher-level languages will be used for development and compiled to Simplicity. Ivy [10] and the Σ-State Authentication Language [11], are existing efforts that may be suitable for being compiled to Simplicity. If these higher-level languages come with formal semantics of their own, we will have the opportunity to prove correct the compiler to Simplicity for these languages. For the time being, generating Simplicity with our Haskell and Coq libraries is possible.

# 6 Conclusion

We have defined a new language designed to be used for crypto-currencies and blockchains. It could be deployed in new blockchain applications, including sidechains [4], or possibly in Bitcoin itself. Simplicity has the potential to be used in any application where finitary programs need to be transported and executed under potentially adversarial conditions.

Our language is bounded, without loops, but is expressive enough to represent any finitary function. These constraints allow for general purpose static

analysis that can effectively bound the amount of computational resources needed by a Simplicity program prior to execution.

Simplicity has simple, functional semantics, which make formal reasoning with software proof assistants relatively easy. This provides the means for people who develop smart contract to formally verify the correctness of their programs.

We have written several low-level functions in Simplicity such as addition, subtraction, and multiplication for various finite-bit words and formally verified their correctness. With these we have built mid-level functions such as elliptic curve addition and multiplication, ECDSA signature validation, and a SHA-256 compression function. This is already sufficient to create simple single and multi-signature verification programs in Simplicity. We have formally verified our SHA-256 compression function is correct and have plans to formally verify the remaining functions.

# A Simplicity Semantics in Coq

Below is the formal definition of core Simplicity as expressed in Coq [29]. The `Term` type is the type of well-typed Simplicity expressions. The `eval` function provides the denotational semantics of well-typed Simplicity expressions.

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

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.

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.

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.
```

# B Full Word Adders in Simplicity

Below we recursively define Simplicity programs for ripple-carry full-adders for any $2^n$-bit word size. A full-adder takes two $n$-bit words and a carry-in bit as inputs and returns a carry-out bit and an $n$-bit word.

$$
\begin{aligned}
&\mathsf{full\text{-}adder}_n : (2^n \times 2^n) \times 2 \vdash 2 \times 2^n \\
&\mathsf{full\text{-}adder}_1 := \mathsf{comp}\,(\mathsf{pair}\,(\mathsf{take}\,\mathsf{half\text{-}adder})\,(\mathsf{drop}\,\mathsf{iden})) \\
&\qquad (\mathsf{comp}\,(\mathsf{pair}\,(\mathsf{take}\,(\mathsf{take}\,\mathsf{iden}))) \\
&\qquad (\mathsf{comp}\ (\mathsf{pair}\,(\mathsf{take}\,(\mathsf{drop}\,\mathsf{iden}))\,(\mathsf{drop}\,\mathsf{iden})) \\
&\qquad\quad \mathsf{half\text{-}adder})) \\
&\qquad (\mathsf{pair}\ (\mathsf{case}\,(\mathsf{injr}\,\mathsf{unit})\,(\mathsf{drop}\,(\mathsf{take}\,\mathsf{iden}))) \\
&\qquad\quad (\mathsf{drop}\,(\mathsf{drop}\,\mathsf{iden}))))) \\
&\mathsf{full\text{-}adder}_{2n} := \mathsf{comp}\,(\mathsf{pair}\,(\mathsf{take}\,(\mathsf{pair}\ (\mathsf{take}\,(\mathsf{take}\,\mathsf{iden})) \\
&\qquad\qquad\qquad\qquad (\mathsf{drop}\,(\mathsf{take}\,\mathsf{iden)))))) \\
&\qquad (\mathsf{comp}\ (\mathsf{pair}\ (\mathsf{take}\,(\mathsf{pair}\ (\mathsf{take}\,(\mathsf{drop}\,\mathsf{iden})) \\
&\qquad\qquad\qquad\qquad\qquad (\mathsf{drop}\,(\mathsf{drop}\,\mathsf{iden))))) \\
&\qquad\qquad\qquad (\mathsf{drop}\,\mathsf{iden})) \\
&\qquad\qquad \mathsf{full\text{-}adder}_n)) \\
&\qquad (\mathsf{comp}\ (\mathsf{pair}\,(\mathsf{drop}\,(\mathsf{drop}\,\mathsf{iden})) \\
&\qquad\qquad (\mathsf{comp}\ (\mathsf{pair}\ (\mathsf{take}\,\mathsf{iden}) \\
&\qquad\qquad\qquad\qquad (\mathsf{drop}\,(\mathsf{take}\,\mathsf{iden))))) \\
&\qquad\qquad\qquad\qquad \mathsf{full\text{-}adder}_n)) \\
&\qquad (\mathsf{pair}\ (\mathsf{drop}\,(\mathsf{take}\,\mathsf{iden})) \\
&\qquad\quad (\mathsf{pair}\,(\mathsf{drop}\,(\mathsf{drop}\,\mathsf{iden}))\,(\mathsf{take}\,\mathsf{iden))))))
\end{aligned}
$$

**Theorem B.1 (Full Adder Correct)** *Let $n$ be a power of two. Let $a, b : 2^n$ be two $n$-bit words. Let $c : 2$ be a bit. Then*

$$\lceil x \rceil_2 \cdot 2^n + \lceil y \rceil_{2^n} = \lceil a \rceil_{2^n} + \lceil b \rceil_{2^n} + \lceil c \rceil_2$$

*where*

$$[\![\mathsf{full\text{-}adder}]\!]\langle\langle a, b\rangle, c\rangle = \langle x, y\rangle$$

We have verified the above theorem in Coq.

We have similarly defined multiplication for $2^n$-bit words and proven its correctness theorem in Coq.

# C Tail Composition Optimization

We define a variant of the translation of Simplicity to the Bit Machine, $\langle\!\langle\cdot\rangle\!\rangle^*$, to add a tail composition optimization (TCO). This optimization moves the **dropFrame** instruction earlier, potentially reducing the memory requirements for execution by the Bit Machine. This is analogous to the tail call optimization found in other languages. Our new definition will be a pair of mutually recursive functions, $\langle\!\langle\cdot\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{off}}$ and $\langle\!\langle\cdot\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{on}}$.

The definition of $\langle\!\langle\cdot\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{off}}$ is identical to that of $\langle\!\langle\cdot\rangle\!\rangle^*$, replacing $\langle\!\langle\cdot\rangle\!\rangle^*$ by $\langle\!\langle\cdot\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{off}}$, except for the $\langle\!\langle \mathsf{comp}\, s\, t : A \vdash C\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{off}}$ clause which is given below.

$$\begin{aligned}\langle\!\langle \mathsf{comp}\, s\, t : A \vdash C\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{off}} \coloneqq\ & \mathbf{newFrame}(\mathsf{bitSize}(B));\\ & \langle\!\langle s : A \vdash B\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{off}}; \mathbf{moveFrame}; \langle\!\langle t : B \vdash C\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{on}}\end{aligned}$$

In the tail position of the comp combinator, we removed the **dropFrame** instruction and call $\langle\!\langle\cdot\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{on}}$ instead. The definition of $\langle\!\langle\cdot\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{on}}$ is given in Figure 4.

We have formally verified in Coq the following correctness theorem for $\langle\!\langle\cdot\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{off}}$.

**Theorem C.1 (Correctness of TCO Operational Semantics)** *Let $t : A \vdash B$ be any Simplicity expression. Let $v : A$ be any value of type $A$. Initialize a Bit Machine with*

- *the value $\ulcorner v \urcorner_A$ as the only frame on the read-frame stack with its cursor at the beginning of the frame, and*
- *the value $[?]^{\mathsf{bitSize}(B)}$ as the only frame on the write-frame stack with its cursor at the beginning of the frame.*

*After executing the instructions $\langle\!\langle t : A \vdash B\rangle\!\rangle^{\mathsf{tco}}_{\mathsf{off}}$, the final state of the Bit Machine has*

- *the value $\ulcorner v \urcorner_A$ as the only frame on the read-frame stack with its cursor at the beginning of the frame, and*
- *the value $\ulcorner [\![t]\!](v) \urcorner_B$ as the only frame on the write-frame stack with its cursor at the end of the frame.*

*In particular, the Bit Machine never crashes during this execution.*

$$
\begin{aligned}
\llbracket \mathsf{iden} : A \vdash A \rrbracket^{\mathsf{tco}}_{\mathsf{on}} &\coloneqq \mathbf{copy}(\mathsf{bitSize}(A)); \mathbf{dropFrame} \\
\llbracket \mathsf{comp}\, s\, t : A \vdash C \rrbracket^{\mathsf{tco}}_{\mathsf{on}} &\coloneqq \mathbf{newFrame}(\mathsf{bitSize}(B)); \llbracket s : A \vdash B \rrbracket^{\mathsf{tco}}_{\mathsf{on}}; \\
&\qquad \mathbf{moveFrame}; \llbracket t : B \vdash C \rrbracket^{\mathsf{tco}}_{\mathsf{on}} \\
\llbracket \mathsf{unit} : A \vdash \mathbb{1} \rrbracket^{\mathsf{tco}}_{\mathsf{on}} &\coloneqq \mathbf{dropFrame} \\
\llbracket \mathsf{injl}\, t : A \vdash B + C \rrbracket^{\mathsf{tco}}_{\mathsf{on}} &\coloneqq \mathbf{write}(\mathsf{0}); \mathbf{skip}(\mathsf{padl}(B, C)); \llbracket t : A \vdash B \rrbracket^{\mathsf{tco}}_{\mathsf{on}} \\
\llbracket \mathsf{injr}\, t : A \vdash B + C \rrbracket^{\mathsf{tco}}_{\mathsf{on}} &\coloneqq \mathbf{write}(\mathsf{1}); \mathbf{skip}(\mathsf{padr}(B, C)); \llbracket t : A \vdash C \rrbracket^{\mathsf{tco}}_{\mathsf{on}} \\
\llbracket \mathsf{case}\, s\, t : (A + B) \times C \vdash D \rrbracket^{\mathsf{tco}}_{\mathsf{on}} &\coloneqq \text{match } \mathbf{read} \text{ with} \\
&\qquad \begin{cases} \mathsf{0} \to \mathbf{fwd}(1 + \mathsf{padl}(A, B)); \\ \qquad \llbracket s : A \times C \vdash D \rrbracket^{\mathsf{tco}}_{\mathsf{on}} \\ \mathsf{1} \to \mathbf{fwd}(1 + \mathsf{padr}(A, B)); \\ \qquad \llbracket t : B \times C \vdash D \rrbracket^{\mathsf{tco}}_{\mathsf{on}} \end{cases} \\
\llbracket \mathsf{pair}\, s\, t : A \vdash B \times C \rrbracket^{\mathsf{tco}}_{\mathsf{on}} &\coloneqq \llbracket s : A \vdash B \rrbracket^{\mathsf{tco}}_{\mathsf{off}}; \llbracket t : A \vdash C \rrbracket^{\mathsf{tco}}_{\mathsf{on}} \\
\llbracket \mathsf{take}\, t : A \times B \vdash C \rrbracket^{\mathsf{tco}}_{\mathsf{on}} &\coloneqq \llbracket t : A \vdash C \rrbracket^{\mathsf{tco}}_{\mathsf{on}} \\
\llbracket \mathsf{drop}\, t : A \times B \vdash C \rrbracket^{\mathsf{tco}}_{\mathsf{on}} &\coloneqq \mathbf{fwd}(\mathsf{bitSize}(A)); \llbracket t : B \vdash C \rrbracket^{\mathsf{tco}}_{\mathsf{on}}
\end{aligned}
$$

Figure 4: Operational semantics for Simplicity using the Bit Machine with TCO.

The proof proceeds by establishing that the machine state transformation induced by executing the instructions

$$\llbracket t : A \vdash B \rrbracket^{\mathsf{tco}}_{\mathsf{off}}; \mathbf{dropFrame}$$

and the instructions

$$\llbracket t : A \vdash B \rrbracket^{\mathsf{tco}}_{\mathsf{on}}$$

are identical.

We would like an improved static analysis of the memory use of this TCO execution. Figure 5 defines the static analysis $\mathsf{cellBnd}^{\mathsf{tco}}(t : A \vdash B)$ which bounds the memory use of the TCO execution. The static analysis becomes somewhat more intricate with the more complex translation. But we may proceed with confidence because we have a formal verification in Coq of the following theorem.

**Theorem C.2 (Static Analysis of Cell Usage with TCO)** *Let $t : A \vdash B$ be any Simplicity expression. Let $v : A$ be any value of type $A$. Initialize a Bit Machine as specified in Theorem C.1. The maximum number of cells in both stacks of the Bit Machine at any point during the execution of $\llbracket t : A \vdash B \rrbracket^{\mathsf{tco}}_{\mathsf{off}}$ from this initial state never exceeds $\mathsf{cellBnd}^{\mathsf{tco}}(t : A \vdash B)$.*

The proof proceeds by establishing that the additional memory used by $\llbracket t : A \vdash B \rrbracket^{\mathsf{tco}}_{\mathsf{off}}$ and $\llbracket t : A \vdash B \rrbracket^{\mathsf{tco}}_{\mathsf{on}}$ in any machine state is no more than $\mathsf{max}(n_1, n_2)$ and $\mathsf{max}(n_1 - m, n_2)$ respectively where $\langle n_1, n_2 \rangle = \mathsf{extraCellBnd}^{\mathsf{tco}}(t :$

$A \vdash B$) and $m$ is the number of cells in the active read frame before executing $\langle\!\langle t : A \vdash B \rangle\!\rangle^{\mathsf{tco}}_{\mathsf{on}}$.

There is a possibility for a head composition optimization where the **newFrame**(**n**) instruction is delayed in order to potentially save memory. It is unclear to us if this is worth the added complexity, so we have not pursued this yet.

## Acknowledgements

Thank you to Shannon Appelcline for his help editing this paper.

$$
\begin{aligned}
\mathsf{extraCellBnd}^{\mathsf{tco}}(\mathsf{iden} : A \vdash A) &\coloneqq \langle 0, 0\rangle \\
\mathsf{extraCellBnd}^{\mathsf{tco}}(\mathsf{comp}\, s\, t : A \vdash C) &\coloneqq \text{let } \langle n_1, n_2\rangle = \\
&\quad \mathsf{extraCellBnd}^{\mathsf{tco}}(s : A \vdash B) \text{ in} \\
&\quad \text{let } \langle m_1, m_2\rangle = \\
&\quad \mathsf{extraCellBnd}^{\mathsf{tco}}(t : B \vdash C) \text{ in} \\
&\quad \text{let } b = \mathsf{bitSize}(B) \text{ in} \\
&\quad \langle \mathsf{max}(b + n_1, m_1, b + m_2), b + n_2\rangle \\
\mathsf{extraCellBnd}^{\mathsf{tco}}(\mathsf{unit} : A \vdash \mathbb{1}) &\coloneqq \langle 0, 0\rangle \\
\mathsf{extraCellBnd}^{\mathsf{tco}}(\mathsf{injl}\, t : A \vdash B + C) &\coloneqq \mathsf{extraCellBnd}^{\mathsf{tco}}(t : A \vdash B) \\
\mathsf{extraCellBnd}^{\mathsf{tco}}(\mathsf{injr}\, t : A \vdash B + C) &\coloneqq \mathsf{extraCellBnd}^{\mathsf{tco}}(t : A \vdash C) \\
\mathsf{extraCellBnd}^{\mathsf{tco}}(\mathsf{case}\, s\, t : (A + B) \times C \vdash D) &\coloneqq \text{let } \langle n_1, n_2\rangle = \\
&\quad \mathsf{extraCellBnd}^{\mathsf{tco}}(s : A \times C \vdash D) \\
&\quad \text{in} \\
&\quad \text{let } \langle m_1, m_2\rangle = \\
&\quad \mathsf{extraCellBnd}^{\mathsf{tco}}(t : B \times C \vdash D) \\
&\quad \text{in} \\
&\quad \langle \mathsf{max}(n_1, m_1), \mathsf{max}(n_2, m_2)\rangle \\
\mathsf{extraCellBnd}^{\mathsf{tco}}(\mathsf{pair}\, s\, t : A \vdash B \times C) &\coloneqq \text{let } \langle n_1, n_2\rangle = \\
&\quad \mathsf{extraCellBnd}^{\mathsf{tco}}(s : A \vdash B) \text{ in} \\
&\quad \text{let } \langle m_1, m_2\rangle = \\
&\quad \mathsf{extraCellBnd}^{\mathsf{tco}}(t : A \vdash C) \text{ in} \\
&\quad \langle m_1, \mathsf{max}(n_1, n_2, m_2)\rangle \\
\mathsf{extraCellBnd}^{\mathsf{tco}}(\mathsf{take}\, t : A \times B \vdash C) &\coloneqq \mathsf{extraCellBnd}^{\mathsf{tco}}(t : A \vdash C) \\
\mathsf{extraCellBnd}^{\mathsf{tco}}(\mathsf{drop}\, t : A \times B \vdash C) &\coloneqq \mathsf{extraCellBnd}^{\mathsf{tco}}(t : B \vdash C)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{cellBnd}^{\mathsf{tco}}(t : A \vdash B) \coloneqq \text{let } \langle n_1, n_2\rangle &= \mathsf{extraCellBnd}^{\mathsf{tco}}(t : A \vdash B) \text{ in} \\
&\mathsf{bitSize}(A) + \mathsf{bitSize}(B) + \mathsf{max}(n_1, n_2)
\end{aligned}
$$

Figure 5: Definition of $\mathsf{cellBnd}^{\mathsf{tco}}$.

# References


[1] G. Andresen. BIP16: Pay to script hash. Bitcoin Improvement Proposal, 2012. https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki.

[2] A. W. Appel. Verification of a cryptographic primitive: '-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, April 2015.

[3] A. W. Appel. Verifiable C. http://vst.cs.princeton.edu/download/VC.pdf, July 2016.

[4] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille. Enabling blockchain innovations with pegged sidechains, 2014. https://www.blockstream.com/sidechains.pdf.

[5] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM.

[6] bitcoinwiki. Script. https://en.bitcoin.it/w/index.php?title=Script&oldid=61707, 2016.

[7] BtcDrak, M. Friedenbach, and E. Lombrozo. BIP112: Checksequenceverify. Bitcoin Improvement Proposal, 2015. https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki.

[8] V. Buterin. CRITICAL UPDATE Re: DAO Vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/, June 2016.

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

[10] Chain. Announcing ivy playground, May 2017. https://blog.chain.com/announcing-ivy-playground-395364675d0a.

[11] A. Chepurnoy. $\sigma$-state authentication language, an alternative to bitcoin script. Poster Session at the 21st International Conference on Financial Cryptography and Data Security, April 2017.

[12] Ethereum. Solidity Documentation: Release 0.4.14. https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf, June 2017.

[13] 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.

[14] S. Goldfeder, J. Bonneau, R. Gennaro, and A. Narayanan. Escrow protocols for cryptocurrencies: How to buy physical goods using bitcoin. `http://fc17.ifca.ai/preproceedings/paper_122.pdf`, April 2017. To appear in the proceedings of the 21st International Conference on Financial Cryptography and Data Security.

[15] Bitcoin Developer Guide. Signature hash types, 2014. `https://bitcoin.org/en/developer-guide#signature-hash-types`.

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

[17] J. Lau. BIP114: Merkelized abstract syntax tree. Bitcoin Improvement Proposal, 2016. `https://github.com/bitcoin/bips/blob/master/bip-0114.mediawiki`.

[18] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[19] G. Maxwell. Zero-knowledge contingent payment, 2011. `https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment`.

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

[21] S. Nakamoto. misc changes. `https://github.com/bitcoin/bitcoin/commit/4bd188c4383d6e614e18f79dc337fbabe8464c82`, August 2010. `https://bitcoin.svn.sourceforge.net/svnroot/bitcoin/trunk@131`.

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

[23] National institute of standards and technology. FIPS pub 186-4 federal information processing standards publication digital signature standard (dss). 2013.

[24] 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, August 2015.

[25] R. O'Connor. Covenants in elements alpha, 2016. Blog post, `https://blockstream.com/2016/11/02/covenants-in-elements-alpha.html`.

[26] Parity. The multi-sig hack: A postmortem, July 2017. `https://blog.parity.io/the-multi-sig-hack-a-postmortem/`.

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

[28] C. P. Schnorr. Efficient identification and signatures for smart cards. In *Proceedings of CRYPTO '89*, 1989.

[29] The Coq Development Team. The Coq Proof Assistant Reference Manual: Version 8.6. `https://coq.inria.fr/refman/`, 2016.

[30] P. Todd. BIP65: Op_checklocktimeverify. Bitcoin Improvement Proposal, 2014. `https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki`.

[31] G. Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. `http://gavwood.com/paper.pdf`.

[32] C. Yarvin, P. Monk, A. Dyudin, and R. Pasco. Urbit: A solid-state interpreter, May 2016. `http://media.urbit.org/whitepaper.pdf`.