# Towards Adding Variety to Simplicity

Nachiappan Valliappan[1], Solène Mirliaz[2], Elisabet Lobo Vesga[1], and Alejandro Russo[1]

[1] Chalmers University of Technology
[2] ENS Rennes

**Abstract.** Simplicity is a Turing incomplete type-combinator language for smart contracts with a formal semantics. The design of Simplicity makes it possible to statically estimate the resources (e.g., memory) required to execute contracts. Such a feature is highly relevant in blockchain applications to efficiently determine fees to run smart contracts. Despite being Turing incomplete, the language is capable of expressing non-trivial contracts. Often, Simplicity programs contain lots of code repetition that could otherwise be avoided if it had common programming languages features, such as local definitions, functions, and bounded loops. In this work, we provide the foundations to make Simplicity a richer language. To achieve that, we connect Simplicity's primitives with a categorical model. By doing so, we lift the language to a more abstract representation that will allow us to extend it by leveraging category theory models for computations. This methodology facilitates the addition of local definitions, functions, and bounded loops. We provide an implementation of Simplicity and its virtual machine in the functional programming language Haskell.

**Keywords:** Blockchain · Smart contracts · Simplicity · Functional Programming · Haskell.

## 1 Introduction

Blockchain technology has emerged as a revolutionary approach for decentralized peer-to-peer networks. The most known deployment of this technology is Bitcoin [5]. Since its launch in 2009, Bitcoin has spawned a number of alternative crypto-currencies using different optimizations and tweaks (e.g., Litecoin, Ripple, EOS [8,9]). Among these, Ethereum [11] stands out for its implementation of programmable transactions in the form of smart contracts. Given that smart contracts are programs, they need to be executed in order to get a result but without compromising the availability of the whole network. To achieve that, Ethereum assigns a consumable resource, called *gas*, to the execution of contracts which is paid by users to the block miners in *ether*—Ethereum's currency [11]. Ethereum uses a Turing-complete computational model, which makes it challenging to predict the gas required to run contracts.

Simplicity [6] is a language for smart contracts with a formal semantics that enables "fast" (linear time) static analysis of resource consumption. The operational semantics of Simplicity instructions is given in an abstract machine named *the Simplicity Bit Machine* (SBM). Despite that the language is capable of expressing non-trivial contracts, it can be very cumbersome to actually write one using its minimal constructs. Moreover,

| Primitive | Description |
|---|---|
| *iden* : $A \vdash A$ | It is the identity function which simply returns its input. |
| *unit* : $A \vdash \mathbb{1}$ | It is a unit function which always returns a value of the unit type. |
| *comp* $f\ g : A \vdash C$ | It composes two simplicity functions $f : A \vdash B$ and $g : B \vdash C$. |
| *pair* $s\ t : A \vdash (B \times C)$ | It constructs a product using $s : A \vdash B$ and $t : A \vdash C$. |
| *take* $t : A \times B \vdash C$ | It applies $t : A \vdash C$ to the first component of a product. |
| *drop* $t : A \times B \vdash C$ | It applies $t : B \vdash C$ to the second component of a product. |
| *injl* $t : A \vdash B + C$ | It constructs a coproduct using $t : A \vdash B$. |
| *injr* $t : A \vdash B + C$ | It constructs a coproduct using $t : A \vdash C$. |
| *case* $s\ t : (A + B) \times C \vdash D$ | It is used to pattern match over the coproduct $(A + B)$ in the input. If the coproduct contains a value of type $A$, then $s : A \times C \vdash D$ is executed, else if the coproduct contains a value of type $B$, then $t : B \times C \vdash D$ is executed. |

Fig. 1: Simplicity's basic functions and combinators

the lack of common programming languages features such as local definitions, functions, and loops forces programs to contain lots of code repetition that could otherwise be avoided.

In this work, we show how to interpret Simplicity as a mathematical model from category theory. Once in the territory of category theory, we borrow its results on modeling different computational aspects to extend Simplicity and its virtual machine with functions. By adding functions, Simplicity contracts can account for local definitions as well as bounded loops. We also provide an implementation of Simplicity, the SBM, as well as our extensions in the functional programming language Haskell [3].

## 2 Background

Simplicity can be considered a typed functional programming language, where the expressions are essentially built from applying the functions in the language. It therefore consists of base functions and function combinators (or combinators for short). Combinators are dedicated to build more complex functions from simpler ones in a compositional manner. Simplicity has three types: the unit type, written $\mathbb{1}$, the product type, written $A \times B$, and the coproduct type, written $A + B$. The entire Simplicity's interface is shown in Figure 1, where $f : i \vdash o$ denotes that the input and output type of function $f$ are $i$ and $o$, respectively. Simplicity's functions are self-explanatory and therefore we omit discussing them further.

One of the design goals for Simplicity is to enable the estimation of runtime resources statically when executed in a virtual machine. The analysis of runtime resources requires a formal model of the runtime as well as an operational semantics of Simplicity's basic functions and combinators. Observe that the computational power of the

[3] Our implementation and accompanying material are available at https://bitbucket.org/russo/isola-additional-material/overview.

language is Turing incomplete (e.g., it lacks loops), which facilitates the estimation of resource consumption—we refer the interested reader to [6] for details.

## 2.1 The Bit Machine

The Simplicity Bit Machine (SBM) is used to execute Simplicity programs and it consists on an state composed of two stacks of data *frames*: the read stack and the write stack. A frame is a list of cells, where each cell contains either **0**, **1** or an undefined value noted as **?**. Each frame has also a cursor, which indicates which cell is to be written or read. The read stack is used to provide the input of the Simplicity function and the write stack is used to write its output. The topmost frame—also called the active frame—contains the input (output) of the current primitive in execution. For instance, in order to execute a Simplicity function $f : A \vdash B$, the active read frame must have a value of type $A$. After execution, the output value of type $B$ can be found on the active write frame.

Simplicity's types have "finite size", that is, well-typed values have a finite representation in terms of cells. In other words, it is always possible to compute the number of cells required by the input and output of well-typed functions. That is, in terms of number of bits, sizeOf($\mathbb{1}$) $= 0$ (as there is only one value), sizeOf($A + B$) $= 1 + \max($sizeOf($A$) $+$ sizeOf($B$)$)$ (where the extra bit is used as a flag to indicate whether the value is of type $A$ or $B$), and sizeOf($A \times B$) $=$ sizeOf($A$) $+$ sizeOf($B$) bits. The ability to compute the size from the types plays a crucial role in the operational semantics of Simplicity. From now on, when referring to the *size of a type*, the reader should keep in mind that we are referring to the representation of values of such a type.

The size of the output type is required to allocate the amount of needed cells for writing the output of a Simplicty function. Similarly, the size of the input type is needed to read the exact number of cells which contain the input. The following outlines show how values of a specific type are read or written in the SBM. Note that all the reading (writing) always happens on the active read (write) frame. Below, we briefly describe how SBM behaves when operating with values of different types. The complete operational semantics of the SBM can be found in [6].

▶ To write a value of type $\mathbb{1}$ on the write frame, the SBM writes nothing (as only one value exists). Similarly, to read a value of type $\mathbb{1}$, the SBM reads nothing.

▶ To write a value of type $A \times B$, the SBM writes the value of type $A$ followed by the value of type $B$ on the write frame. Instead, to read a value of $A \times B$, the SBM first computes the size of $A$ and reads that many cells in order to get $A$. Then, it computes the size of $B$ and reads that many cells in order to get $B$.

▶ To write a value of type $A + B$, the SBM writes a (cell) flag bit indicating whether the value is $A$ (0) or $B$ (1). After that, it skips any excess cells which may have been allocated (keeping in mind that the value could be $A$ or $B$), and then writes the available value. This mechanism of skipping ahead is also called *padding*.

Since the resource allocation in the read and write stack is made using the type information (as shown above), and given that the language is Turing incomplete, it becomes possible to do static analysis to compute an upper bound on the runtime resources used by an smart contract. For example, it is possible to estimate the number of cells used by a Simplicity program on both stacks. We refer the reader to [6] for a detailed discussion on static analysis in Simplicity programs.

## 3 Categorical Semantics for Simplicity

In this section, we establish an unforeseen connection between Simplicity and a branch of mathematics called category theory. Such connection will open the door to apply known results from category theory [3] in order to systematically extend Simplicity and the SBM with new features. We start by briefly describing a specific kind of category: the Bi-Cartesian Categories, or BCCs for short. Then, we show how categories can be used to model Simplicity computations.

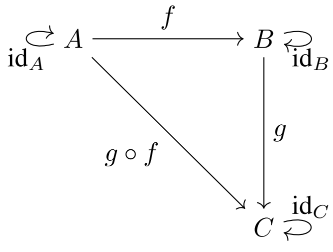
$$\begin{tikzcd} A \arrow[loop left, "\mathrm{id}_A"] \arrow[r, "f"] \arrow[dr, "g \circ f"'] & B \arrow[loop right, "\mathrm{id}_B"] \arrow[d, "g"] \\ & C \arrow[loop right, "\mathrm{id}_C"] \end{tikzcd}$$

Fig. 2: Identity and composition in a category

A *category* is composed of objects and morphisms between these objects. A simple way to think about it is to consider it as a *graph with certain operations and satisfying certain properties*, where the vertices are the objects and the (oriented) edges the morphisms. Category theory will often characterize the features of the categories, based on the relations between objects and morphisms.

The basic features that a category must have are identity and composition.

► *Identity* For every object $A$ in the category (i.e., every vertex in the graph), there exists an identity morphism (edge) from $A$ to $A$, noted $\mathrm{id} : A \to A$. Since there are many identity morphisms, it is common to identify them by their associated objects, e.g., $\mathrm{id} : A \to A$ is denoted by $\mathrm{id}_A$. For simplicity, while presenting the construction of some categorical features as graphs, we often omit the identity morphisms but recall that they do exist for every object (vertex).

► *Composition* For every two morphisms (edges) $f : A \to B$ and $g : B \to C$, there exists a morphism (edge) $g \circ f :: A \to C$. Furthermore, the composition must be associative, and the morphism id must be the identity for composition, which gives the following equalities: $f \circ (g \circ h) = (f \circ g) \circ h$ and that $f \circ \mathrm{id}_A = \mathrm{id}_B \circ f = f$.

To give an example of a category, let us consider three objects, namely $A$, $B$, and $C$, and two morphisms $f : A \to B$ and $g : B \to C$. If we want to place them into a category, we must add an identity morphism for each object and a morphism for the composition of $f$ and $g$. Figure 2 shows the structure of such a category.

The rest of the section proceeds to describe the remaining features found in BCCs.

► *Terminal object* There is an object, we noted it as $T$, such that for any other object $A$ in the category, there exists precisely one morphism $\mathrm{final} : A \to T$ (also know as terminal morphism). Figure 3a) shows objects $A$ and $B$ and their corresponding morphisms to the terminal object.

► *Products* For all objects $A$ and $B$ in the category, there exists the product object $A \times B$. Every product object comes equipped with two morphisms $\mathrm{fst} : A \times B \to A$

$$\begin{tikzcd}
A \arrow[loop left, "\mathrm{id}_A"] \arrow[r, "\mathrm{final}_A"] & T \arrow[loop below, "\mathrm{id}_T"] & B \arrow[l, "\mathrm{final}_B"'] \arrow[loop right, "\mathrm{id}_B"]
\end{tikzcd}$$

(a) Terminal object and morphisms

$$\begin{tikzcd}
 & C \arrow[d, dashed, "\langle f, g\rangle"] \arrow[ddl, bend right, "f"'] \arrow[ddr, bend left, "g"] & \\
 & A \times B \arrow[dl, "\mathrm{fst}"] \arrow[dr, "\mathrm{snd}"'] & \\
A & & B
\end{tikzcd}$$

(b) Products

Fig. 3: Terminal and products in BCCs

and snd : $A \times B \rightarrow B$ which project out its components. Importantly, for every two morphisms $f : C \rightarrow A$ and $g : C \rightarrow B$, there exists an unique morphism (represented by a dashed arrow) called *factor*, written $\langle f, g\rangle : C \rightarrow A \times B$, which should fulfill the following equations: $f = \mathrm{fst} \circ \langle f, g\rangle$ and that $g = \mathrm{snd} \circ \langle f, g\rangle$. These equations capture the behavior of factor, i.e., a product element obtained from $C$ is constructed by building an element $A$ with $f$ and an element $B$ with $g$. Figure 3b) introduces objects $A$, $B$, $C$, morphisms $f : C \rightarrow A$ and $g : C \rightarrow B$, and describes their relation via the product object $A \times B$ and the factor morphism.

► *Coproducts* For all objects $A$ and $B$ in the category, there exists a coproduct object $A + B$. Every coproduct object comes with two morphims, the injections $\mathrm{inj}_1 : A \rightarrow A + B$ and $\mathrm{inj}_2 : B \rightarrow A + B$. If we have two morphisms $f : (E \times A) \rightarrow C$ and $g : (E \times B) \rightarrow C$, then there exists a unique morphism called *copair*, written $[f, g] : (E \times (A + B)) \rightarrow C$ [4]. This morphism fulfills the equations: $f = [f, g] \circ \langle \mathrm{id}_E, \mathrm{inj}_1\rangle$ and $g = [f, g] \circ \langle \mathrm{id}_E, \mathrm{inj}_2\rangle$. In other words, the copair builds an element of $C$ by using either $f$ or $g$, depending on either it receives an element of $A$ or $B$. Figure 4 introduces objects $A$, $B$, $C$ and $E$, morphisms $f : E \times A \rightarrow C$ and $g : E \times B \rightarrow C$, and describes their relation via the coproduct object $A + B$ and the copair morphism.

$$\begin{tikzcd}
A \arrow[dr, "\mathrm{inj}_1"] & & B \arrow[dl, "\mathrm{inj}_2"'] \\
 & A + B &
\end{tikzcd}$$

(a) Coproducts as dual of products

$$\begin{tikzcd}
E \times A \arrow[ddr, bend right, "f"'] & & E \times B \arrow[ddl, bend left, "g"] \\
 & E \times (A + B) \arrow[d, dashed, "{[f, g]}"] & \\
 & C &
\end{tikzcd}$$

(b) Copair

Fig. 4: Coproducts in BCCs

[4] In category theory, *copair* is commonly used without the product with $E$: if $f' : A \rightarrow C$ and $g' : B \rightarrow C$, then $[f', g'] : A + B \rightarrow C$. However, using the morphism containing $E$ will ease the equivalence between morphisms and Simplicity terms.

## 3.1 Simplicity and BCCs

The type signature of BCCs' morphisms and Simplicity's basic and combinator functions look pretty similar. In this section, we describe how to model Simplicity functions using BCCs. Intuitively, the idea is that a function $f : A \vdash B$ will be modeled by a morphism $m : A \to B$. In other words, Simplicity types become objects in BCCs and functions morphisms. For instance, the function *iden* $: A \vdash A$ can be modeled by the morphism $\mathrm{id} : A \to A$. The complete translation of Simplicity to BCCs is given on Figure 5, where we denote $f \rightsquigarrow m$ as the relation "the morphism $m$ models the Simplicity function $f$".

| Simplicity | | BCCs |
|---|---|---|
| $\mathit{iden} : A \vdash A$ | $\rightsquigarrow$ | $\mathrm{id}_A : A \to A$ |
| $\mathit{comp}\ (s : A \vdash B)\ (t : B \vdash C) : A \vdash C$ | $\rightsquigarrow$ | $g \circ f : A \to C$<br>where $s \rightsquigarrow f : A \to B$<br>$t \rightsquigarrow g : B \to C$ |
| $\mathit{unit} : A \vdash \mathbb{1}$ | $\rightsquigarrow$ | $\mathrm{final} : A \to T$ |
| $\mathit{injl}\ (t : A \vdash B) : A \vdash B + C$ | $\rightsquigarrow$ | $(\mathrm{inj}_1 : B \to B + C) \circ g : A \to (B + C)$<br>where $t \rightsquigarrow g : A \to B$ |
| $\mathit{injr}\ (t : A \vdash C) : A \vdash B + C$ | $\rightsquigarrow$ | $(\mathrm{inj}_2 : C \to B + C) \circ f : A \to (B + C)$<br>where $t \rightsquigarrow f : A \to C$ |
| $\mathit{case}\ (s : A \times C \vdash D)$<br>$(t : B \times C \vdash D) : (A + B) \times C \vdash D$ | $\rightsquigarrow$ | $[f \circ \mathit{flip}, g \circ \mathit{flip}] \circ \mathit{flip} : (A + B) \times C \to D$<br>where $s \rightsquigarrow f : A \times C \to D$<br>$t \rightsquigarrow g : B \times C \to D$<br>$\mathit{flip} = \langle \mathrm{snd}, \mathrm{fst} \rangle$ |
| $\mathit{pair}\ (s : A \vdash B)\ (t : A \vdash C) : A \vdash B \times C$ | $\rightsquigarrow$ | $\langle f, g \rangle : A \to (B \times C)$<br>where $s \rightsquigarrow f : A \to B$<br>$t \rightsquigarrow g : A \to C$ |
| $\mathit{take}\ (t : A \vdash C) : A \times B \vdash C$ | $\rightsquigarrow$ | $f \circ (\mathrm{fst} : A \times B \to A) : A \times B \to C$<br>where $t \rightsquigarrow f : A \to C$ |
| $\mathit{drop}\ (t : B \vdash C) : A \times B \vdash C$ | $\rightsquigarrow$ | $f \circ (\mathrm{snd} : A \times B \to B) : A \times B \to C$<br>where $t \rightsquigarrow f : B \to C$ |

Fig. 5: Translation from Simplicity terms to BCCs morphims

The most interesting case is the translation of *case* $s\ t$. While *case* has type $(A + B) \times C \vdash D$, its closest morphism—copair—has type $(C \times (A + B)) \to D$, hence we cannot use it directly since the type signatures do not align. From category theory, however, we know about the symmetry of products, i.e., $A \times B$ and $B \times A$ are provably isomorphic and therefore there must exist an isomorphism between them. We use one direction of that isomophism—called *flip* in Figure 5—to build the corresponding morphism of *case*.

By mapping Simplicity functions into BCCs, the attentive reader could be afraid that we might be introducing or restricting the behavior of Simplicity programs. For example, on one hand, products need to fulfill certain equations in BCCs (recall previous Section). On the other hand, there is no relation stated for Simplicity operators like *pair*, *take*, and *drop*. It is easy to show that Simplicity operators already fulfill all the equations required by BCCs. We refer readers to the accompanying material for the details of the proof.

We have now established the connection between Simplicity functions and BCCs morphisms and we can start adding more features to Simplicity (Section 5). Category theory will guide us toward the implementation of user-defined functions. This would be a significant improvement to Simplicity, as it would allow to write simpler and shorter programs.

## 4 Implementation

In this section, we present another of our contributions: an implementation of Simplicity, its categorical model, and the SBM as embedded domain-specific languages (eDSL) in Haskell [4]. To implement BCCs in Haskell, we need to determine what the objects and morphisms are going to be in Haskell. By doing so, we restrict ourselves to a particular class of BCCs that we call $\text{BCCs}_{\text{Hask}}$, where categorical objects are represented with Haskell types.

```
data T
data a :∗: b
data a :+: b
```

Fig. 6: Simplicity Types

We model both types in Simplicity and objects in $\text{BCCs}_{\text{Hask}}$ with the empty types given in Figure 6. Type $T$ is the unit/terminal, type $a\ {:}{*}{:}\ b$ is the product, and type $a\ {:}{+}{:}\ b$ is the coproduct. In what follows, we will model the term language of Simplicity and morphisms in $\text{BCCs}_{\text{Hask}}$ using *Generalized Algebraic Data Types* (GADTs) [7]. The use of GADTs allows us to directly encode the typing judgements of Simplicity and $\text{BCCs}_{\text{Hask}}$ in the constructors. In that manner, the Haskell's type checker ensures that Simplicity functions and $\text{BCCs}_{\text{Hask}}$ morphisms are well-typed *by construction*.

### 4.1 An eDSL for Simplicity

We model Simplicity programs as values of the following GADT parameterized over an input type $i$ and an output type $o$:

```
data Simpl i o where
  Iden  :: SType a ⇒ Simpl a a
  Unit  :: SType a ⇒ Simpl a T
  Take  :: (SType a, SType b, SType c) ⇒ Simpl a c → Simpl (a :∗: b) c
  Drop  :: (SType a, SType b, SType c) ⇒ Simpl b c → Simpl (a :∗: b) c
  Injl  :: (SType a, SType b, SType c) ⇒ Simpl a b → Simpl a (b :+: c)
  Injr  :: (SType a, SType b, SType c) ⇒ Simpl a c → Simpl a (b :+: c)
  Comp :: (SType a, SType b, SType c) ⇒
            Simpl a b → Simpl b c → Simpl a c
  Pair  :: (SType a, SType b, SType c) ⇒
            Simpl a b → Simpl a c → Simpl a (b :∗: c)
```

```
Case :: (SType a, SType b, SType c, SType d) ⇒ Simpl (a :∗: c) d →
          Simpl (b :∗: c) d → Simpl ((a :+: b) :∗: c) d
```

The type constraint $\mathit{SType}\ a$ restricts the domain over which the type variable $a$ ranges over. In our case, a type variable $a$ satisfies the constraint $\mathit{SType}\ a$ only if it is instantiated with $T$, $a\ :\!*\!:\ b$ or $a\ :\!+\!:\ b$, where $a$ and $b$ are simplicity types themselves. The reason for adding this constraint is two fold: first, to ensure that a Simplicity expression cannot be created for some arbitrary Haskell type such as $[\mathit{Int}]$ (as this might break the property that the size of the type can be determined statically), and second, to implement a function $\mathit{sizeOf}$ to calculate the size (in bits) of a Simplicity type—which is used later to run programs on the SBM.

In Haskell, type constraint $\mathit{SType}$ is implemented as a type class, and the Simplicity types which satisfy it are implemented as instances of such a class:

```
class SType a where
  sizeOf :: a → Int
instance SType T where
  ...
instance (SType a, SType b) ⇒ SType (a :+: b) where
  ...
instance (SType a, SType b) ⇒ SType (a :∗: b) where
  ...
```

(Ellipsis are used to denote Haskell code that is not relevant for the point being made.) Each Simplicity type instance must provide a definition for the $\mathit{sizeOf}$ function. Recall that the SBM works by allocating cells in the stack frames based on the type information (Section 2.1). For brevity, we skip the implementation of $\mathit{sizeOf}$ but it can be found in the accompanying material. Later in Section 4.4, we show how to leverage $\mathit{sizeOf}$ to implement the SBM.

### 4.2 An eDSL for BCCs$_{\text{Hask}}$

In BCCs$_{\text{Hask}}$, we model objects as Haskell types and morphisms as values of the GADT $\mathit{Mph}$:

```
data Mph obj a b where
  Id       :: obj a ⇒ Mph obj a a
  Terminal :: obj a ⇒ Mph obj a T
  Fst      :: (obj a, obj b) ⇒ Mph obj (a :∗: b) a
  Snd      :: (obj b, obj b) ⇒ Mph obj (a :∗: b) b
  Inj1     :: (obj a, obj b) ⇒ Mph obj a (a :+: b)
  Inj2     :: (obj a, obj b) ⇒ Mph obj b (a :+: b)
  ⊙        :: (obj a, obj b, obj c) ⇒
                Mph obj b c → Mph obj a b → Mph obj a c
  Factor   :: (obj a, obj b1, obj b2) ⇒
                Mph obj a b1 → Mph obj a b2 → Mph obj a (b1 :∗: b2)
```

$$
\begin{aligned}
&\mathit{CoFactor} :: (\mathit{obj}\ a, \mathit{obj}\ b, \mathit{obj}\ c, \mathit{obj}\ e) \Rightarrow \mathit{Mph}\ \mathit{obj}\ (e :\!*\!: a)\ c \rightarrow \\
&\qquad\qquad \mathit{Mph}\ \mathit{obj}\ (e :\!*\!: b)\ c \rightarrow \mathit{Mph}\ \mathit{obj}\ (e :\!*\!: (a :\!+\!: b))\ c
\end{aligned}
$$

This data type is parameterized over a type constraint $\mathit{obj}$ and objects $a$ and $b$. Each constructor of this data type constructs a morphism in a given $\text{BCC}_{\text{Hask}}$. A type constraint $\mathit{obj}\ a$ ensures the type $a$ is indeed an object of the considered $\text{BCC}_{\text{Hask}}$, and not some arbitrary Haskell type.

The main difference between $\mathit{SType}$ in $\mathit{Simpl}$ and $\mathit{obj}$ in $\mathit{Mph}$ is that $\mathit{SType}$ is a specific type constraint, while $\mathit{obj}$ is parameterized over. Observe that different instantiations of $\mathit{obj}$ might encode different $\text{BCCs}_{\text{Hask}}$. For instance, if $\mathit{obj}$ gets instantiated with $\mathit{SType}$, we obtain a $\text{BCC}_{\text{Hask}}$ which models Simplicity in Haskell (as shown in the next Section) [5]. From now on, we refer to this category as simply $\text{BCC}_{\text{Hask}}$.

### 4.3 A Translation from Simplicity to $\text{BCC}_{\text{Hask}}$

The translation from Simplicity to $\text{BCC}_{\text{Hask}}$ is a Haskell function (named $\mathit{simpl2mph}$) between the eDSLs presented above. In other words, we show how to translate a program $\mathit{prog} :: \mathit{Simpl}\ i\ o$ to a morphism $m :: \mathit{Mph}\ \mathit{SType}\ i\ o$. The constraint $\mathit{obj}$ is now instantiated with $\mathit{SType}$, and hence the objects in the $\text{BCC}_{\text{Hask}}$ are Simplicity types. The translation is essentially a syntactic translation of the rules in Figure 5—a nice aspect of our approach.

$$
\begin{aligned}
&\mathit{simpl2mph} :: \mathit{Simpl}\ i\ o \rightarrow \mathit{Mph}\ \mathit{SType}\ i\ o \\
&\mathit{simpl2mph}\ \mathit{Iden} &&= \mathit{Id} \\
&\mathit{simpl2mph}\ \mathit{Unit} &&= \mathit{Terminal} \\
&\mathit{simpl2mph}\ (\mathit{Take}\ f) &&= \mathit{simpl2mph}\ f \odot \mathit{Fst} \\
&\mathit{simpl2mph}\ (\mathit{Drop}\ f) &&= \mathit{simpl2mph}\ f \odot \mathit{Snd} \\
&\mathit{simpl2mph}\ (\mathit{Injl}\ f) &&= \mathit{Inj}_1 \odot (\mathit{simpl2mph}\ f) \\
&\mathit{simpl2mph}\ (\mathit{Injr}\ f) &&= \mathit{Inj}_2 \odot (\mathit{simpl2mph}\ f) \\
&\mathit{simpl2mph}\ (\mathit{Pair}\ p\ q) &&= \mathit{Factor}\ (\mathit{simpl2mph}\ p)\ (\mathit{simpl2mph}\ q) \\
&\mathit{simpl2mph}\ (\mathit{Comp}\ f\ g) &&= \mathit{simpl2mph}\ g \odot \mathit{simpl2mph}\ f \\
&\mathit{simpl2mph}\ (\mathit{Case}\ p\ q) &&= (\mathit{CoFactor}\ (\mathit{simpl2mph}\ p \odot \mathit{flip}) \\
& &&\qquad\qquad\qquad (\mathit{simpl2mph}\ q \odot \mathit{flip})) \odot \mathit{flip} \\
&\quad \textbf{where} \\
&\quad\quad \mathit{flip} = \mathit{Factor}\ \mathit{Snd}\ \mathit{Fst}
\end{aligned}
$$

As explained in Section 3, constructor $\mathit{Case}\ p\ q$ needs an auxiliary morphism $\mathit{flip}$ to use $\mathit{CoFactor}$.

### 4.4 The SBM

Given the close correspondence between Simplicity's primitives and BCCs' morphisms, the execution of morphisms on the SBM is very similar to the execution of Simplicity

[5] The encoding of a category using the eDSL for $\text{BCCs}_{\text{Hask}}$ does not ensure that the category is indeed a BCC. It is the programmers responsibility to ensure this by verifying the existence of constructed morphisms and proving the corresponding laws. The eDSL is simply the "language of BCCs where objects are Haskell types."

functions. A given morphism is translated to instructions of the SBM, which are then executed on the SBM to yield the output.

We start by looking at the SBM interface. The instructions of the SBM are implemented as a Haskell data type (see Figure 7a). For brevity, we only show some of the instructions here. Type $Bit$ is an alias for $Bool$ representing a single bit value on the SBM.

```
data Inst = Nop
   | Write Bit
   | Copy Int
   | Skip Int
   | Fwd Int
   | Read
   ...
```

(a) SBM Instructions

```
1 type Frame = ([Maybe Bit], Int)
2 type Stack  = [Frame]
3 data Machine = Machine
4    {readStack  :: Stack
5    , writeStack :: Stack}
6 type SBM = State Machine
```

(b) SBM components

Fig. 7: SBM data types

A list of these instructions are run on the SBM using the function:

$$run :: [Inst] \rightarrow SBM\ (Maybe\ Bit)$$

where output type $SBM$ is a monadic type [10] which encapsulates the stateful behavior of the SBM. This design choice arise from noticing that the evaluation of each instruction may change the state of the SBM, and hence affect the execution of subsequent instructions. More specifically, the $SBM$ type is defined as shown in Figure 7b line 6, where a value of type $Machine$ (lines 3-5) represents a configuration of the virtual machine at a given moment. The configuration is composed of read ($readStack$) and write ($writeStack$) stacks, which are themselves composed of frames. A frame is a list of cells paired with a cursor. The cursor points to the current cell in the frame and is implemented as an $Int$ representing the index of the current cell. A cell is encoded as a $Maybe\ Bit$, as it can host an undefined value (recall Section 2.1). A cell with an undefined value is represented by $Nothing$, otherwise it is a $Just$ value with a $Bit$.

A given $\text{BCC}_{\text{Hask}}$ morphism is translated into a list of SBM instuctions using the function

$$mph2sbm :: Mph\ Types\ a\ b \rightarrow [Inst]$$

We will look at a few cases of the $mph2sbm$ implementation to illustrate how it works. To understand how to map a morphism $m : A \rightarrow B$ into the SBM, we need to think of it as a Simplicity function $f : A \vdash B$ (recall that we proved that such models are equivalent in Section 3.1). In this light, the instructions corresponding to $m$ must assume (before their execution) that the machine is initialized with a configuration where a value of type $A$ is on the active read frame. Post execution of $m$, the active write frame must contain a value of type $B$. For example, to execute the morphism $\mathrm{id} : A \rightarrow A$, the value of $A$ must be available on the read stack. The expected end configuration is the same value of $A$ on the write stack. That is, we need to copy as many bits as the size of $A$ from the read stack to the write stack. This operation is achieved by using the $Copy$ instruction. To determine the size of $A$, we use the $sizeOf$ function—where the constraint $SType$ (introduced earlier) on type $A$ comes into action. The implementation of this case is as follows:

$$mph2sbm\ (Id :: Mph\ SType\ a\ a) = [\,Copy\ (sizeOf\ (\bot :: a))]$$

(Observe that this definition works for any identity morphism since it is polymorphic in $a$). To give $sizeOf$ an argument of type $a$, we must construct a value of that type. For this, we use the value $\bot$ which constructs (or inhabits) every Haskell (and hence Simplicity) type. Notice that Simplicity types are empty data types, and the inhabitant of the type has no significance. We are only interested in the type $a$ as it gives us the corresponding definition of $sizeOf$.

```
mph2sbm ((g :: Mph SType b c) ⊙
  (f :: Mph SType a b)) =
  [NewFrame (sizeOf (⊥ :: b))]
   ++ mph2sbm f
   ++ [MoveFrame]
   ++ mph2sbm g
   ++ [DropFrame]
```

Fig. 8: Implementation of $\odot$

We implement composition as show in Figure 8. We first allocate memory for the intermediate result of type $b$, run $f$ (which writes the intermediate result on the active write frame), move the active write frame to the read stack (using $MoveFrame$), and finally run $g$, which writes the result of type $c$ on the active write frame; having at the end the expected configuration after executing $(\odot)$. Since the intermediate result of type $b$ is no longer needed, we drop the active read frame (using $DropFrame$). Implementing the compilation of the other morphisms is analogous and can be found in the accompanying material.

## 5 Adding functions to Simplicity

In this section, we extend the Simplicity core language with user-defined functions, provide categorical semantics for the extension, and also extend the evaluation model (SBM) to support the extended semantics. To achieve this, we leverage the exposed connection between Simplicity and categorical models (recall Section 3). From the latter, we use the concept of *exponential* objects as a guideline to model functions. We briefly introduce what it means for a category to have exponentials and discuss their relation to functions in Simplicity.

► *Exponentials* For objects $B$ and $C$ in a category, an exponential object is a special object (denoted as $B \Rightarrow C$), for which there exists a morphism $\mathrm{eval} : (B \Rightarrow C) \times B \to C$. Additionally, for every morphism $f : A \times B \to C$, there must exist a unique morphism $\mathrm{curry}(f) : A \to B \Rightarrow C$ such that $f = \mathrm{eval} \circ \langle \mathrm{curry}(f) \circ \mathrm{fst}, \mathrm{snd}\rangle$. That is, in a category with exponentials, for every morphism $f : A \times B \to C$, there exists a *curried* version of it, i.e., $\mathrm{curry}(f)$. Figure 9 shows $f$ and the morphisms involving exponentials—namely $\mathrm{curry}(f)$ and eval.

An exponential object is the categorical generalization of the *function type* ($\rightarrow$). Operation curry generalizes the construction of a *lambda abstraction*—also known as currying in lambda calculus [1]. The eval morphism generalizes the *application* of a function of type $B \rightarrow C$ to an argument of type $B$ to return a value of type $C$.

$$\begin{tikzcd}
A \times B \arrow[r, "f"] \arrow[d, "\langle \text{curry}(f) \circ \text{fst}, \text{snd} \rangle"'] & C \\
(B \Rightarrow C) \times B \arrow[ur, "\text{eval}"'] &
\end{tikzcd}$$

Fig. 9: Exponentials in BCCCs

```
data Simpl i o where
  ...
  Lam :: (SType a, SType b, SType c) ⇒
           Simpl (a :∗: b) c → Simpl a (b :⇒: c)
  App :: (SType a, SType b, SType c) ⇒
           Simpl a (b :⇒: c) → Simpl a b → Simpl a c
```

(a) Functions in Simplicity

```
data Mph obj a b where
  ...
  Curry :: (obj a, obj b, obj c) ⇒
             Mph obj (a :∗: b) c → Mph obj a (b :⇒: c)
  Eval  :: (obj b, obj c) ⇒ Mph obj ((b :⇒: c) :∗: b) c
```

(b) Exponentials in BCC

Fig. 10: Implementation of Functions and Exponentials

Exponential objects are implemented by the following data type:

```
data a :⇒: b
```

which represents the exponential object $a \Rightarrow b$ for some objects $a$ and $b$. To add the new morphisms, we extend *Mph* with the new constructors *Curry* and *Eval* (see Figure 10b) as described in Figure 9. When we include exponentials in a BCC, it becomes a Bi-Cartesian Closed Category or a BCCC.

In Simplicity, $a$ :⇒: $b$ is a function type which expects an argument of type $a$ and returns a value of type $b$ (where $a$ and $b$ are Simplicity types). We add new primitives to Simplicity's eDSL as shown in Figure 10a. The constructor *Lam* accepts a Simplicity term whose input and output types are $(a$ :∗: $b)$ and $c$ respectively, and constructs a new term *Simpl* $a$ $(b$ :⇒: $c)$—where the input is a value of type $a$ and the output is a function of type $b$ :⇒: $c$. The *App* constructor, on the other hand, accepts a Simplicity term which returns a function of type $b$ :⇒: $c$ and another term which returns a value of type $b$, and constructs a term which returns a value of type $c$.

The translation of the newly added Simplicity terms to $\text{BCCC}_{\text{Hask}}$ (i.e., BCCCs where objects are Haskell types) is defined as follows:

$$\begin{aligned} &\mathit{simpl2mph}\ (\mathit{Lam}\ f) &&= \mathit{Curry}\ (\mathit{simpl2mph}\ f) \\ &\mathit{simpl2mph}\ (\mathit{App}\ f\ x) &&= \mathit{Eval}\ \odot\ (\mathit{Factor}\ (\mathit{simpl2mph}\ f)\ (\mathit{simpl2mph}\ x)) \end{aligned}$$

This translation provides the categorical semantics for functions in Simplicity, and hence forms the basis for implementing them.

## 5.1 Using functions in Simplicity

Note that the language extension in the previous section does not just allow for functions to be defined, but also treats functions as values. This allows for programming with higher order functions and facilitates some powerful abstractions. For example, functions can be used to introduce *let-bindings* into the language. Let-bindings greatly reduce the duplication of sub-expressions in the language. In the presence of functions, they can be easily encoded using function application as $(\mathbf{let}\ x = e\ \mathbf{in}\ e') = (\lambda x \rightarrow e')\ e$.

Another example of the usefulness of functions is the ability to define a loop combinator. The $\mathit{loop}$ combinator (defined below) can be used to repetitively apply a Simplicity term to an input value. Term $\mathit{loop}\ f\ n$ applies $f$ on the input $n$ times. This is possible only when $f$ has the same input and output type, and is hence expected to have the type $\mathit{Simpl}\ a\ a$. Symbol $n$ is a Simplicity term of type $\mathit{SNat}$ (defined below) which encodes a natural number using using just function abstraction and application—known as *Church numerals* in lambda calculus.

$$\begin{aligned} &\mathbf{type}\ \mathit{SNat} = \forall a.\ \mathit{Types}\ a \Rightarrow \mathit{Simpl}\ (a\ :\!\Rightarrow\!:\ a)\ (a\ :\!\Rightarrow\!:\ a) \\ &\mathit{loop} :: \mathit{Types}\ a \Rightarrow \mathit{Simpl}\ a\ a \rightarrow \mathit{SNat} \rightarrow \mathit{Simpl}\ a\ a \\ &\mathit{loop}\ f\ n = \mathit{App}\ (\mathit{App}\ (\mathit{toLam}\ n)\ (\mathit{toLam}\ f))\ \mathit{Iden} \\ &\quad \mathbf{where} \\ &\qquad \mathit{toLam} :: (\mathit{Types}\ a, \mathit{Types}\ b, \mathit{Types}\ r) \Rightarrow \mathit{Simpl}\ a\ b \rightarrow \mathit{Simpl}\ r\ (a\ :\!\Rightarrow\!:\ b) \\ &\qquad \mathit{toLam}\ s = \mathit{Lam}\ (\mathit{Drop}\ s) \end{aligned}$$

$$\begin{aligned} &\mathit{zero} :: \mathit{SNat} \\ &\mathit{zero} = \mathit{Lam}\ (\mathit{Drop}\ \mathit{Iden}) \\ &\mathit{one} :: \mathit{SNat} \\ &\mathit{one} = \mathit{Lam}\ (\mathit{App}\ (\mathit{Take}\ \mathit{Iden}) \\ &\qquad (\mathit{Drop}\ \mathit{Iden})) \end{aligned}$$

Fig. 11: Church numerals

For the Haskell aware reader, note that we use higher-ranked types to define $\mathit{SNat}$—a feature of the Haskell type system which is not available in Simplicity. While this might appear disconcerting, note that this is not a strict requirement to define a $\mathit{loop}$ combinator. We could instead encode $\mathit{SNat}$ as $\mathit{SNat}\ a$, removing the explicit quantification (∀) and hence the need for higher-ranked types.

Since the programmer must provide a construction of a Simplicity term of type $\mathit{SNat}$ (which always represents a finite number), the loop can only be used for a finite number of iterations. Figure 11 illustrates the construction of some of such natural numbers.

## 5.2 Implementing functions on SBM

In this section, we extend the SBM—the primary evaluation model of the Simplicity language—to support higher order functions. To do this, we must implement the translation of $Curry$ and $Eval$ morphisms to SBM instructions. We start by requiring that an exponential object $a \;{:}{\Rightarrow}{:}\; b$ must also be a valid Simplicity type that satisfies the $SType$ constraint. Consequently, we must implement an instance of the type class $SType$ for the type $a \;{:}{\Rightarrow}{:}\; b$, i.e., we need to provide a definition for $sizeOf\ (a \;{:}{\Rightarrow}{:}\; b)$. For that, we need to identify a way to store and retrieve exponential objects in the SBM.

Notice that the serialization of a morphism captured in the exponential $a \;{:}{\Rightarrow}{:}\; b$ can be arbitrary long, as the morphisms can be arbitrary complex. As a result, it is not possible to know the number of bits needed to serialize such morphisms only by looking at the type $a \;{:}{\Rightarrow}{:}\; b$. This is problematic since the SBM is not meant to manipulate types with arbitrary sizes.

To address this issue, we extend the SBM with a new field responsible to store a list of exponentials. We then represent exponentials in the stack frames as merely pointers (indexes) into such list. We have not yet defined the size of pointers, but we assume them to occupy the amount of bits given by a parameter $sizePtr$—we will see later how to statically compute it. Additionally, we must devise new SBM instructions responsible to execute the $Curry$ and $Eval$ morphisms, i.e., instructions responsible to create and apply exponentials.

If we follow the philosophy of Simplicity that the input (output) type should indicate the values to be read (write) into the stack, a morphism of the form $Curry\ f :: Mph\ SType\ a\ (b \;{:}{\Rightarrow}{:}\; c)$ must be compiled to an instruction that reads a value of type $a$ from the read stack and places the exponential (of type $b \;{:}{\Rightarrow}{:}\; c$) in the write frame. In this light, we introduce the instruction $PutClosure$ responsible to allocate exponentials:

```
mph2sbm :: Mph SType a b → Int → [Inst]
mph2sbm (Curry (f :: Mph Types (a :∗: b) c)) sizePtr =
  let aSize = sizeOf (⊥ :: a) sizePtr
  in [PutClosure (mph2sbm f sizePtr) aSize sizePtr]
```

Observe that $mph2sbm$ takes the size of pointers as an extra argument as well as $sizeOf$—note that $sizeOf$ could be called on a pointer and thus it needs to know its size ($sizeOf\ (a \;{:}{\Rightarrow}{:}\; b)\ sizePtr = sizePtr$). The instruction $PutClosure$ takes tree arguments: the compilation of the curried morphism $f :: Mph\ \ Types\ \ (a : * : b)\ \ c$ ($mph2sbm\ f\ \ sizePtr$), the amount of bits to be read from the read stack ($aSize$), and the size of pointers ($sizePtr$). When the SBM executes this instruction, it allocates an exponential as the pair composed of $f$'s instructions, paired with the value of type $a$ read from the stack—this semantics is inspired by how Cousineau et al. handle exponentials in the Categorical Abstract Machine [2] as closures. The output in the write stack of $PutClosure$ is the pointer to the recently allocated exponential. For instance, Figure 12 illustrates the effect of running an instruction $PutClosure\ \ [Read; ...]\ \ 2\ \ 4$ under a given configuration of the machine.

In the same line of reasoning, morphism $Eval :: Mph\ SType\ ((b \;{:}{\Rightarrow}{:}\; c) :\!*\!: b)\ c$ should be compiled to an instruction which reads an exponential (i.e., a pointer) together with a value of type $b$ from the read stack and produces a $c$ in the write stack. To achieve that, we introduce the instruction $EvalClosure$ in charge of using the exponentials:

```
mph2sbm (Eval :: Mph SType ((b :⇒: c) :∗: b) c) sizePtr =
  [EvalClosure sizePtr (sizeOf (⊥ :: b) sizePtr)]
```

| Read Stack | Write Stack | List of exponentials |
|---|---|---|
| $[\ldots\underset{\uparrow}{1}1\ \ldots]$ | $[\ldots\underset{\uparrow}{?}????\ldots]$ | $0000\ ([\mathit{NewFrame}, \mathit{Write}\ 1, \ldots], [0010])$ |
| $[\ldots]$ | $[\ldots]$ | |

$$\Big\downarrow \mathit{run}\ (\mathit{PutClosure}\ [\mathit{Read}, \ldots]\ 2\ 4)$$

| Read Stack | Write Stack | Closures list |
|---|---|---|
| $[\ldots\underset{\uparrow}{1}1\ \ldots]$ | $[\ldots\ 0001\underset{\uparrow}{?}\ldots]$ | $0000\ ([\mathit{NewFrame}, \mathit{Write}\ 1, \ldots], [0010])$ |
| $[\ldots]$ | $[\ldots]$ | $0001\ ([\mathit{Read}, \ldots], [11])$ |

Fig. 12: Executing $\mathit{PutClosure}$ in the SBM

This instruction takes the size of a pointer ($\mathit{sizePtr}$) together with the size of the value of type $b$ ($\mathit{sizeOf}\ (\bot :: b)\ \mathit{sizePtr}$). When executed, $\mathit{EvalClosure}$ fetches the exponential via the pointer, and places the value of type $b$ obtained from the read stack as an input to the instructions that constitute the exponential. (There are actually many intermediate steps to reach that configuration and we refer the interested reader to the accompanying material for details.) After the instructions of the exponential get executed, the machine will have a value of type $c$ in the active write frame.

We still need to define $\mathit{sizeOf}$ for the pointers manipulated in the stacks. To know the maximal size (in bits) to encode a pointer, we must know the maximal number of closures existing in a Simplicity program. This number is actually the amount of $\mathit{Curry}$ occurrences in the morphism denoting our program. The reader can convince herself that computing this number is a linear traversal in the size of the morphism. Let $cc$ be the number of $\mathit{Curry}$ in the morphism, then the maximal size (in bits) of the pointers is $\mathit{sizePtr} = \log_2\ (cc) + 1$. Once $\mathit{sizePtr}$ is determined, we can do our translation to SBM instructions by calling $\mathit{mph2sbm}$ with a morphism and $\mathit{sizePtr}$ as a parameter.

## 5.3 Static Analysis

A notable property of Simplicity is the ability to statically estimate computational resources needed by a program. This is achieved using the underlying evaluation model, i.e., the SBM. In this section, we discuss this property in light of the extensions made to Simplicity and the SBM.

In our model, a given Simplicity program is translated to a $\mathrm{BCC}_{\mathrm{Hask}}$ morphism using $\mathit{simpl2mph}$, which is then translated to SBM instructions using $\mathit{mph2sbm}$. Consider the problem of estimating the number of instructions executed by the SBM for a given program. In the absence of exponentials, to count the number of instructions, we simply count the number of instructions returned by $\mathit{mph2sbm}$. However, this straightforward approach fails to hold in the presence of exponentials. This is because the instruction $\mathit{EvalClosure}$ (introduced for the evaluation of exponentials), cannot be treated as a single instruction. $\mathit{EvalClosure}$ contains a pointer to a list of instructions executed by the SBM, which means that it causes several other instructions (including itself) to be executed.

To mitigate this problem, we must also count the number of instructions that are referred to by a pointer of $\mathit{EvalClosure}$. This can be easily calculated in linear time by maintaining an environment which contains the pointers and their corresponding list of instructions as introduced by $PutClosure$.

The static analysis of cell usage described in [6] extends naturally to exponentials since all exponential objects are of a fixed sized $sizePtr$ (discussed in the previous section). However, since our storage model has been extended with a list of closures, we must also estimate the maximum size of the closure list. It should be possible to compute an upper bound on the size of the closure list in linear time by maintaining an external environment (as suggested above). Note that since our extensions do not provide a mechanism to define recursion, such as a fix-point combinator, an attempt to perform static analysis in such a fashion must always terminate. However, we have not implemented this static analysis, and leave it as a suggestion for future work.

## 6 Final remarks

This work provides a new semantics for Simplicity based on category theory, and extends Simplicity with user defined and higher order functions. Using functions, we have established the foundational and practical basis to enrich the language towards other interesting features such as bounded loops. As long as we stay under a computational model similar to the simply typed lambda calculus, we argue that it is possible to carry out "quick" static analysis to predict resource usage in Simplicity programs. We evaluate our theory by providing an implementation of our results and approach in Haskell. Our hope is to make the language even more useful to develop smart contracts with formal guarantees.

## References

1. Barendregt, H., Dekkers, W., Statman, R.: Lambda calculus with types. Cambridge University Press (2013)
2. Cousineau, G., Curien, P., Mauny, M.: The categorical abstract machine. Sci. Comput. Program. **8**(2), 173–202 (1987)
3. Elliott, C.: Compiling to categories. Proc. ACM Programing Languages **1**(ICFP) (Sep 2017). https://doi.org/http://dx.doi.org/10.1145/3110271, http://conal.net/papers/compiling-to-categories
4. Marlow, S., et al.: Haskell 2010 language report. Available online http://www. haskell. org/(May 2011) (2010)
5. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
6. OConnor, R.: Simplicity: A new language for blockchains. arXiv preprint arXiv:1711.03028 (2017)
7. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for gadts. In: ACM SIGPLAN Notices. vol. 41, pp. 50–61. ACM (2006)
8. Schwartz, D., Youngs, N., Britto, A., et al.: The ripple protocol consensus algorithm. Ripple Labs Inc White Paper **5** (2014)
9. Swan, M.: Blockchain: Blueprint for a new economy. " O'Reilly Media, Inc." (2015)
10. Wadler, P.: Monads for functional programming. In: International School on Advanced Functional Programming. pp. 24–52. Springer (1995)
11. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151**, 1–32 (2014)