FUNDAÇÃO GETÚLIO VARGAS

MODELAGEM MATEMÁTICA

A Simplified version of Bitcoin, implemented in Agda

Student:
Guilherme Horta Alvares da
Silva
Professor:
Flávio Codeço Coelho

Abstract

A cryptocurrency is a digital currency that works in a decentralized way, without a central authority and its states are maintained through distributed consensus. It has an important role in society because it is money that is ruled only by algorithms and it avoids big central power, like banks and government.

Agda is a functional programming language with dependent types. It is also a proof assistant based on the preposition-as-types paradigm, like Coq. This language is useful to prove properties about the code.

We present in this work an explanation about what is cryptocurrencies and their principal characteristics, a brief explanation about Lambda Calculus, dependents types and Agda, and we present a cryptocurrency model made in this language. Most of all parts of Bitcoin are coded and typed in this model. Since transactions, transactions tree, ledger, block, and blockchain. Cryptographic functions are all postulated like hash functions, transformation functions of a private key into a public key and addresses. Besides, in this work, there is code that transforms and validates transactions from plain text into our model.

Keywords: verification; formal methods; Agda; smart contract; Blockchain; cryptocurrency; Bitcoin

Resumo

Uma criptomoeda é uma moeda digital que funciona de maneira descentralizada, sem uma autoridade central e seus estados são mantidos por meio de consenso distribuído. Ela tem um papel importante na sociedade, porque é um dinheiro que é governado apenas por algoritmos e evita grande centralização de poder, como a de bancos e a do governo.

Agda é uma linguagem de programação funcional com tipos dependentes. É também um assistente de prova baseado no paradigma de preposição como tipos, assim como Coq. Essa linguagem é útil para provar propriedades sobre o código.

Apresentamos nesse trabalho uma explicação sobre o que são criptomoedas e suas principais características, uma breve explicação sobre o Lambda Calculus, tipos de dependentes e Agda, e apresentamos um modelo de criptomoeda feito nessa linguagem. A maioria das partes do Bitcoin é codificada e programada nesse modelo. Desde transações, árvore de transações, *Ledger*, bloco e cadeia de blocos. As funções criptográficas, como funções hash, funções de transformação de uma chave privada em uma chave pública e seus endereços, são postuladas. Além disso, neste trabalho, há código que transforma e valida transações de um texto sem formatação para o nosso modelo.

Acknowledgment

I would like to give thanks to all the people that helped me in this work.

Thanks, Philip Wadler, for taught me about programming language foundations and how to code in Agda. Thanks, Alexandre Rademaker for teaching me the basics of Logic in Lean and the type theory for the Lambda Calculus in this thesis. Thanks, Flávio Coelho for being my advisor, for teaching me what Bitcoin is and helping me writing my dissertation. In the last, thanks Thais Horta (my mother) for helping me writing better English and fixing my grammar mistakes.

Acronyms

CPU Central processing unit. 7

DAO Decentralized Autonomous Organization. 8, 9, 10

EVM Ethereum Virtual Machine. 9, 28

ID Identity. 8

IOHK Input Output HK. 59

PoW proof-of-work. 7

RSA Rivest Shamir Adleman. 7

UTXO Unspent transaction output. 7, 26

zk-SNARK Zero-Knowledge Succinct Non-Interactive Argument of Knowledge. 8

List of Figures

Contents

1   Introduction

1.1   History of cryptocurrencies

In 1983, David Chaum created ecash (Panurach, 1996) an anonymous cryptographic electronic money. This cryptocurrency use Rivest Shamir Adleman (RSA) blind signatures (Chaum, 1983) to spend transactions. Later, in 1989, David Chaum found an electronic money corporation called DigiCash Inc. It was declared bankruptcy in 1998.

Adam Back developed a proof-of-work (PoW) scheme for spam control, Hashcash (Back et al., 2002). To send an email, the hash of the content of this email plus a nonce has to have a numerical value smaller than a defined target. So, to create a valid email, the sender (miner) has to spend a considerable Central processing unit (CPU) resource on it. Miners should calculate a lot of hash values before finding a valid one. This idea is used in Bitcoin proof of work because each block has a nonce guessed by the miner and the hash of the block has to be less than the target value.

Wei Dai proposes b-money (Dai, 1998) for the first proposal for distributed digital scarcity. And Hal Finney created Bit Gold (Wallace, 2011), a reusable proof of work for hash cash for its algorithm of proof of work.

On 31 October 2008, Satoshi Nakamoto registered the website "bitcoin.org" and put a link for his paper (Nakamoto et al., 2008) in a cryptography mailing list. In January 2009, Nakamoto released the Bitcoin software as an open-source code. The identity of Satoshi Nakamoto is still unknown. Since that time, the total market of Bitcoin came to 330 billion dollars in 17 of December of 2018 when its value reached a historic peak of 20 thousand dollars.

Other cryptocurrencies like Ethereum (Wood et al., 2014), Monero (Noether, 2015) and ZCash (Hopwood, Bowe, Hornby, & Wilcox, 2016) were created after Bitcoin, but Bitcoin is still the cryptocurrency with the biggest market value.

Ethereum is a cryptocurrency that uses an account model instead of Unspent transaction output (UTXO) used in Bitcoin for its transaction data structure. It uses Solidity as its programming language for smart contracts which resembles Javascript, so it is easier to program in it than in the stack machine programming language of Bitcoin. Ethereum changed from proof of work (used in Bitcoin) to proof of stake which is now the default proof mechanism of Ethereum 2.0.

Monero and ZCash are both cryptocurrencies that focus on fungibility, privacy, and decentralization. Monero uses an obfuscated public ledger, so anyone can send transactions, but nobody can tell the source, amount or destination. ZCash uses

the concept of zero-knowledge proof called Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK), which guarantees privacy for its users.

1.2   Objectives

Cryptocurrencies are used as money and used in smart contracts in a decentralized way. Because of that, it is not possible to revert a transaction or undo the creation of the smart contract. There is no legal framework or agent to solve a problem in case of the existence of a bug. Because of that, the formal proofs are desirable in the cryptocurrency protocol. So it can avoid big financial loss.

In the case of Bitcoin, if there is some problem in the source code, it is possible to fix it using soft or hard forks. In soft fork, there is an upgrade in the software that is compatible with the old software. So it is possible the existence of old and new nodes in the same Bitcoin network. In hard forks, all the nodes should be upgraded at the same time. Because the newer version is not compatible with the older one. So it is very dangerous to do this kind of fork. Therefore in Bitcoin, this kind of fork just happened twice by accident. It happened in 2013, because of the BerkeleyDB issue and it was solved in BIP 0050. And happened in 2018, because of Denial of Service and inflation vulnerability bug.

For example, in Bitcoin, the uniqueness of transaction Identity (ID) was not guaranteed. To fix this problem, it should put the block number in the coinbase transaction. This kind of change was solved in a soft fork named SegWit.

In Ethereum, there was a bug in Decentralized Autonomous Organization (DAO) smart contract. Malicious users exploited a vulnerability in it with total loss of this exploit was 150 million dollars. There was a hard fork to undo most of the transactions that exploited this contract. This kind of hard fork violates the principle that smart contracts should be ruled just by algorithms without any human intervention. The old Ethereum blockchain that has not done the fork became the Ethereum Classic. It is the version of Ethereum that has never done a hard fork before.

The objective of this work is to give a formal definition of what a cryptocurrency should be. There are some different definitions of a cryptocurrency in this work, but there are some formal proofs that they are the same.

Given that cryptocurrencies and smart contracts should have guarantees of their correctness, it is very important to verify their protocols. The creation of formal methods for Bitcoin protocol and the proof of its correctness are the objective of this work.

So, we are proposing a model of Bitcoin using an interactive theorem prover and dependently typed programming language Agda. We include a formalization of transactions, transactions tree, blocks, blockchain and how messages are signed. In this work, we also include functions that transform raw transactions (made with simple types like lists, naturals) to transactions with dependent types.

1.3   Related Work

Beukema (Beukema, 2014) was one of the first to try to define a formal specification of Bitcoin. In this works, functions interfaces of Bitcoin and what they should do were defined. Most of these functions define how the Bitcoin Network protocol should be. In this work, he does not utilize any programming language with dependent types like Agda or CoQ. mCRL2, a specification programming language, was used.

Chaudhary and his team (Chaudhary, Fehnker, Van De Pol, & Stoelinga, 2015) have created a model of Bitcoin blockchain in the model checker UPPAAL. In his work, they calculate the probability of a malicious attack to succeed in doing a double spend. For a small number of blocks, it is easier to do this attack. Because of that, it is usually recommended that the user waits for more blocks confirmations after a big transaction.

Bastiaan (Bastiaan, 2015) showed a stochastic model of Bitcoin using continuous Markov chains. In his work, he proposes a way of avoiding a 51% attack in the network, using two-phase proof of work.

Orestis Melkonian (Melkonian, 2019) in his masters have done the formal specification of BitML (smart contract language) in Agda. BitML can be compiled to Script, the smart contract language of Bitcoin.

Kosba (Kosba, Miller, Shi, Wen, & Papamanthou, 2016) in his work made a programming language called Hawk for smart contracts. This language uses formal methods to verify privacy using zero-knowledge proofs. Using this language, the programmer does not have to worry about implementing the cryptography, because the compiler generates automatically an effective one.

Bhargavan (Bhargavan et al., 2016) translated Solidity and Ethereum bytecode into F*. He verified that the Ethereum DAO bug was caught in its translation. Nowadays, they have an implementation of Ethereum Virtual Machine (EVM) and Solidity in OCaml, but they want to have a full implementation of EVM in F* too.

Luu (Luu, Chu, Olickel, Saxena, & Hobor, 2016) built a symbolic execution tool named OYENTE to look for potential bugs. In his work, he found a lot of contracts

with real bugs. One of these bugs was the DAO bug, that caused a loss of 60 million dollars. He used Z3 to find a potentially dangerous path of code.

Anton Setzer (Setzer, 2018) also contributed to modeling Bitcoin. He coded in Agda the definitions of transactions and transactions tree of Bitcoin. Orestis Melkonian starts to formalize Bitcoin Script.

My work tries to extend the Anton Setzer model and makes it possible to use the Bitcoin protocol from inputs and outputs from plain text. For example, the user sends a transaction in plain text to the software and it validates if it is correct. To use the Anton Setzer model, the user has to send the data and the proof that are both valid.

2   Relevant Background

2.1   Agda Introduction

Agda is a dependently typed functional language developed by Norell at the Chalmers University of Technology as his Ph.D. Thesis. The current version of Agda is Agda 2.

2.1.1   Syntax

In Agda, *Set* is equal to type. In languages with dependent types, it is possible to create a function that returns a type.

```
bool→Set : (b : Bool) → Set
bool→Set b = if b then ℕ else Bool
```

After the function name, it is two colons (:) and the arguments of the function. It is closed by *(name_of_argument : type_of_argument)*. After all, there is one arrow and the type of the result of the function. This "if, then, else" is not a function built-in in Agda. It is a function defined this way *if_then_else_* .

So it is possible to use this function in the default way.

```
bool→Set-und : Bool → Set
bool→Set-und b = if_then_else_ b ℕ Bool
```

Or use the arguments inside the underscore.

```
bool→Set' : Bool → Set
bool→Set' b = if b then ℕ else Bool
```

The same notation can be done using just arrows without naming the arguments.

Because of dependent types, it is possible to have a type that depends on the input.

It is possible in Agda to do the pattern match. So it breaks the input in cases.

```
boolean→Set : (b : Boolean) → Set
boolean→Set true = ℕ
boolean→Set false = Bool
```

To create a new type with a different pattern match, it is used the data constructor.

```
data Boolean : Set where
  true : Boolean
  false : Boolean
```

This is another example of *Data Set*, but it depends on the argument.

```
data Vec : ℕ → Set where
  [] : Vec zero
  _::_ : {size : ℕ} → ℕ → Vec size → Vec (suc size)

nil : Vec zero
nil = []

vec-one : Vec (suc zero)
vec-one = zero :: nil
```

Vector zero is a type of a vector of size zero, so the only option to construct it is the empty vector. It is constructed from the first constructor. Other types of vectors like *Vector 1* (vector of size one), *Vector 2, ...* can only be constructed by the second constructor. It takes as argument a natural number and a vector and returns a vector with the size of the last vector plus one.

Records are data types with just one case of pattern match.

```
record Person : Set where
  constructor person
  field
    name : String
    age : ℕ

agePerson : (person : Person) → ℕ
agePerson (person name age) = age
```

The constructor is the name of the data constructor.

Implicit terms are elements that the compiler is smart enough to deduce it. So it is not necessary to put it as an argument of the function.

```
id : {A : Set} (x : A) → A
id x = x
```

Implicit arguments are inside *{}*. In this example, the name of the Set (A) can not be omitted (like the second function version of boolean to set), because it is used to say that x is of type A.

In the case of the function *id*, the type of input can be deduced by the compiler. For example, *id* function should be a function from Natural to Natural because *zeroN* is already of type Natural. In this example, the compiler could figure out the term A (Natural) of type *Set*.

```
zeroℕ : ℕ
zeroℕ = id zero
```

Functions in Agda can be defined in two ways

```
id-nat : ℕ → ℕ
id-nat x = x

id-nat' : ℕ → ℕ
id-nat' = λ x → x
```

In the first case, the arguments are before equal sign ($=$). In the second case, it is used the lambda abstraction that means the same thing.

```
case_of_ : ∀ {a b} {A : Set a} {B : Set b} → A → (A → B) → B
case x of f = f x


filter : {A : Set} → (A → Boolean) → List A → List A
filter p [] = []
filter p (x :: xs) =
  case p x of
  λ { true → x :: filter p xs
    ; false → filter p xs
    }
```

Filter is a function that filters all elements that have some property already defined. In this definition of filter, the lambda abstraction can pattern match in the two possibilities of $p\ x$, that are *true* or *false*

```
filter' : {A : Set} → (A → Boolean) → List A → List A
filter' p [] = []
filter' p (x :: xs) with p x
```

```
...  |  true = x :: filter' p xs
...  |  false = filter p xs
```

The *with* abstraction is the same thing as the *case of*. Before the =, there are the possibilities of *p x* (*true* or *false*).

```
postulate someBot : ⊥
```

In Agda, the *postulate* assumes that there is one element of a given type without defining it. In this example, *someBot* is an element of the empty type. Postulate can be dangerous because it assumes that some element of a given type exists, but it can not (like in the example).

```
postulate P : ℕ → Set
postulate plus-commute : (a b : ℕ) → a + b ≡ b + a

thm : (a b : ℕ) → P (a + b) → P (b + a)
thm a b t rewrite plus-commute a b = t
```

In Agda, *rewrite* means changing the returned type of the result. Because $a + b \equiv b + a$, returning an element of the type $P(a + b)$ is the same thing that returning an element of the type $P(b + a)$.

2.1.2 Lambda Calculus

Lambda Calculus is a minimalist Turing complete programming language with the concept of abstraction, application using binding and substitution. For example, x is a variable, $(\lambda x.M)$ is an Abstraction and $(M\ N)$ is an Application.

In Lambda Calculus, there are two types of computational conversions α-conversion and β-reduction. In α-conversion, $(\lambda x.M[x]) \to (\lambda y.M[y])$. So in every free variable x in M will be renamed to y. For $M[x] = x$, an α-conversion is $(\lambda x.x) \to (\lambda y.y)$

A free variable is every variable that is not bound outside. For example, $((\lambda x.x)x)$. The blue x is bonded for the green x, but the red x is not bonded for any function. So the red x is a free variable.

In β-reduction, it replaces the all free for the expression in the application. The β-reduction of this expression $((\lambda x.M)N) \to (M[x := N])$. So if $M = x$, the β-reduction will be $((\lambda x.x)N) \to N$. If $M = (\lambda x.x)x$, the β-reduction will be $(\lambda x.((\lambda x.x)x))N \to (\lambda x.x)N$.

Agda uses typed lambda calculus. So in an application *(M N)*, *M* has to be of type $A \Rightarrow B$ and N has to be of type A. $(\lambda(x : A).x)$ is of type $A \Rightarrow A$, because x is of type A.

id : $\{A : \mathsf{Set}\} \to A \to A$
id $= \lambda\ x \to x$

The simplest function is the identity function made in Agda.

id' : $\{A : \mathsf{Set}\} \to A \to A$
id' $x = x$

This is another way of writing the same function.

true : $\{A : \mathsf{Set}\} \to A \to A \to A$
true $x\ y = x$

false : $\{A : \mathsf{Set}\} \to A \to A \to A$
false $x\ y = y$

This is how true and false are encoded in lambda calculus.

zero : $\{A : \mathsf{Set}\} \to (A \to A) \to A \to A$
zero $suc\ z = z$

one : $\{A : \mathsf{Set}\} \to (A \to A) \to A \to A$
one $suc\ z = suc\ z$

two : $\{A : \mathsf{Set}\} \to (A \to A) \to A \to A$
two $suc\ z = suc\ (suc\ z)$

This is how naturals numbers are defined in lambda calculus. Look that the definition of zero looks like the definition of false.

isZero : $\{A : \mathsf{Set}\} \to ((A \to A) \to A \to A) \to (A \to A \to A)$
isZero $n\ true\ false = n\ (\lambda\ \_ \to false)\ true$

isZero-zero : $\{A : \mathsf{Set}\} \to \mathsf{Result}\ (\mathsf{isZero}\ \{A\}\ \mathsf{zero})$
isZero-zero $= \mathsf{res}\ (\lambda\ true\ false \to true)$

isZero-two : $\{A : \mathsf{Set}\} \to \mathsf{Result}\ (\mathsf{isZero}\ \{A\}\ \mathsf{two})$
isZero-two $= \mathsf{res}\ (\lambda\ true\ false \to false)$

Defining natural numbers in this way, it is possible to say if a natural number is zero or not.

```
plus : {A : Set} → ((A → A) → A → A)
    → ((A → A) → A → A)
    → ((A → A) → A → A)
plus n m = λ suc z → n suc (m suc z)
```

```
_+_ : {A : Set} → ((A → A) → A → A)
    → ((A → A) → A → A)
    → ((A → A) → A → A)
_+_ n m suc z = n suc (m suc z)
```

Plus is defined this way using lambda calculus.

```
one+one : {A : Set} → Result (_+_ {A} one one)
one+one = res (λ suc z → suc (suc z))
```

This is one example of the calculation of one plus one in Lambda Calculus.

```
emptyList : {A List : Set}
    → (A → List → List) → List → List
emptyList _::_ nil = nil
```

```
natList : {A List : Set}
    → (((A → A) → A → A) → List → List) → List → List
natList _::_ nil = one :: (two :: nil)
```

This is how lists are defined in Lambda Calculus.

```
sumList : {A List : Set}
    → Result (natList {A} {(A → A) → A → A} _+_ zero)
sumList = res (λ suc z → suc (suc (suc z)))
```

Substituting the cons operation of list per plus and nil list to zero, it is possible to calculate the sum of the list.

```
left : {A B C : Set} → A → (A → C) → (B → C) → C
left x f g = f x
```

```
right : {A B C : Set} → B → (A → C) → (B → C) → C
right x f g = g x
```

In this way, it is possible to define *Either*. It is one way to create a type that can be a Natural or a Boolean.

```
zero-left : ∀ {A B C}
    → (((A → A) → A → A) → C) → (B → C) → C
zero-left = left zero

one-left : ∀ {A B C}
    → (((A → A) → A → A) → C) → (B → C) → C
one-left = left one

false-right : ∀ {A B C}
    → (A → C) → ((B → B → B) → C) → C
false-right = right false

true-right : ∀ {A B C}
    → (A → C) → ((B → B → B) → C) → C
true-right = right true
```

In these examples, it is defined zero, one in left and false, true in right.

```
zero-isZero : ∀ {A}
    → Result (zero-left {A} isZero id)
zero-isZero = res (λ true false → true)

one-isZero : ∀ {A}
    → Result (one-left {A} isZero id)
one-isZero = res (λ true false → false)

false-id : ∀ {A}
    → Result (false-right {(A → A) → A → A} isZero id)
false-id = res (λ true false → false)

true-id : ∀ {A}
    → Result (true-right {(A → A) → A → A} isZero id)
true-id = res (λ true false → true)
```

Either is useful when defining one function that works for left and another that works for the right. If the natural number is zero, the function chosen is the left and if it is an identity, the function chosen is the right.

```
tuple : {A B C : Set} → A → B → (A → B → C) → C
tuple x y f = f x y
```

This way is how tuple is defined in Lambda Calculus.

```
zero-false : {A B C : Set} → (((A → A) → A → A)
    → (B → B → B) → C) → C
zero-false = tuple zero false

one-true : {A B C : Set} → (((A → A) → A → A)
    → (B → B → B) → C) → C
one-true = tuple one true
```

This is how is defined the tuple zero false and the tuple one true.

```
add-true : {A : Set} → ((A → A) → A → A)
    → (A → A → A) → ((A → A) → A → A)
add-true n b suc z = b (suc (n suc z)) (n suc z)

add-zero-false : {A : Set}
    → Result (zero-false {(A → A) → A → A} add-true)
add-zero-false = res (λ suc z → z)

add-one-true : ∀ {A}
    → Result (one-true {(A → A) → A → A} add-true)
add-one-true = res (λ suc z → suc (suc z))
```

This is one way of defining a function that adds one to the argument if the first element of the tuple is true.

2.1.3   Martin-Löf type theory

Agda also provides proof assistants based on the intentional Martin-Löf type theory.

In Martin-Löf type theory, there are three finite types and five constructors types. The zero type contains zero terms. It is called the empty type and it is written ⊥.

```
data ⊥ : Set where

⊥-elim : {A : Set} (bot : ⊥) → A
⊥-elim ()
```

The first type is the type with just one canonical term and it represents existence. It is called unit type and it is written top.

```
data ⊤ : Set where
  tt : ⊤
```

The second type contains two canonical terms. It represents a choice between two values.

```
data Either {l : Level} (A : Set l) (B : Set l) : Set l where
  left : (l : A) → Either A B
  right : (r : B) → Either A B

Either-elim : {l l2 : Level} {A B : Set l}
  {motive : (eab : Either A B) → Set l2}
  (target : Either A B)
  (on-left : (l : A) → (motive (left l)))
  (on-right : (r : B) → (motive (right r)))
  ------------------------------------------
  → motive target
Either-elim (left l) onleft onright = onleft l
Either-elim (right r) onleft onright = onright r
```

The Boolean type is defined using the Trivial type and the Either type.

```
Bool : Set
Bool = Either ⊤ ⊤
```

If statement is defined using booleans.

```
if_then_else_ : {l : Level} {A : Set l}
  (b : Bool) (tRes fRes : A) → A
if b then tRes else fRes =
  Either-elim b (λ _ → tRes) λ _ → fRes
```

2.1.4   Types Constructors

The product-types contain an ordered pair. The second type can depend on the first type.

```
data ∑ (A : Set) (B : A → Set) : Set where
  ⟨_,_⟩ : (x : A) → B x → ∑ A B
```

```
∑-elim : ∀ {A : Set} {B : A → Set} {C : Set}
   → (∀ x → B x → C)
   → ∑ A B
   ----------------
   → C
∑-elim f ⟨ x , y ⟩ = f x y
```

```
∃-vec : {A : Set} → ∑ ℕ (λ n → Vector A n)
∃-vec = ⟨ zero , [] ⟩
```

In the function $\exists - vec$, the product-type has the same meaning of exists. This type means that "Exists a natural number n that there is one element of type *Vector A n*"

The π-types contain functions. So given an input type, it will return an output type. It has the same meaning as a function:

```
∀-elim : ∀ {A : Set} {B : A → Set}
   (L : ∀ (x : A) → B x)
   (M : A)
   ------------------
   → B M
∀-elim L M = L M
```

In Inductive types, it is a self-referential type. Naturals numbers are examples of that:

```
data ℕ : Set where
   zero : ℕ
   suc : ℕ → ℕ
```

Other data structures like a linked list of natural numbers, trees, graphs are inductive types too.

Proofs in inductive types are made by induction.

```
ℕ-elim : (target : ℕ) (motive : (ℕ → Set))
   (base : motive zero)
   (step : (n : ℕ) → motive n → motive (suc n) )
   → motive target
ℕ-elim zero motive base step = base
```

```
ℕ-elim (suc target) motive base step =
    step target (ℕ-elim target motive base step)
```

Universe types are created to allow proofs written in all types. For example, the type of *Nat* is *U0*.

Agda has dependent types like in CoQ but does not have tactics. Agda is a total language, so it is guaranteed that the code always finish (impossible infinite loops) and coverage all inputs.

Agda has inductive data types that are similar to algebraic data types in non-dependently typed programming languages. The definition of Peano numbers in Agda is:

```
data ℕ : Set where
    zero : ℕ
    suc : ℕ → ℕ
```

Definitions in Agda are done using induction. For example, the sum of two numbers in Agda:

```
_+'_ : ℕ → ℕ → ℕ
zero +' m = m
suc n +' m = suc (n + m)
```

In Agda, because of dependent types, it is possible to make more expressive types that are not possible in other languages. For example, get the first element of a vector. For it, it is necessary to specify in the type that the vector should have a size greater than zero.

```
head : {A : Set} {n : ℕ} (vec : Vector A (suc n)) → A
head (x :: vec) = x
```

Another good example is that in the sum of two matrices, they should have the same dimensions.

```
_+m_ : {m n : ℕ} (P Q : Matrix ℕ m n) → Matrix ℕ m n
[] +m [] = []
(vx :: P) +m (vy :: Q) = (vx +v vy) :: (P +m Q)
```

2.1.5   Preposition as Types

In Agda, types can be seen as prepositions and terms of this given type like a proof. For example, x can be a term of the type A. So x is a proof that A holds.

```
id : {A : Set} (x : A) → A
id x = x
```

In the function *id*, if there is a proof x that A holds, so A holds. The function *id* is a proof of $A \to A$.

```
_and_ : {A B : Set} → A → B → A × B
a and b = a , b
```

In the function *and*, a is a proof that A holds and b is a proof that B holds, so a , b is a proof of $A \times B$

```
eq : ∀ {A : Set} (x : A) → x ≡ x
eq x = refl {_} {_} {x}
```

Equality function can be seen as "For all elements of x in the Set A, $x \equiv x$". If there is a term of this type, so this preposition is true. In this case, the term is a function that given a term x of type A, it returns *refl {_} {_} {x}*. *refl* is a constructor of equality, so *refl {_} {_} {x}* returns a term of type $x \equiv x$.

2.2   Bitcoin Overview

The Bitcoin was made to be a peer to peer electronic cash. It was made in one way that users can save and verify transactions without the need of a trusted party. Because of that no authority or government can shut down the Bitcoin.

Transactions in Bitcoins (like in Figure 1) are an array of input of previous transactions and an array of outputs. Each input and output is an address, each address is made from a public key that is made from a private key.

A private key is a big number. It is so big that it is almost impossible to generate two identical private keys.

The public key is generated from the private key (like in Figure 2 where the account number is f(p)), but a private key can not be generated from a public key.
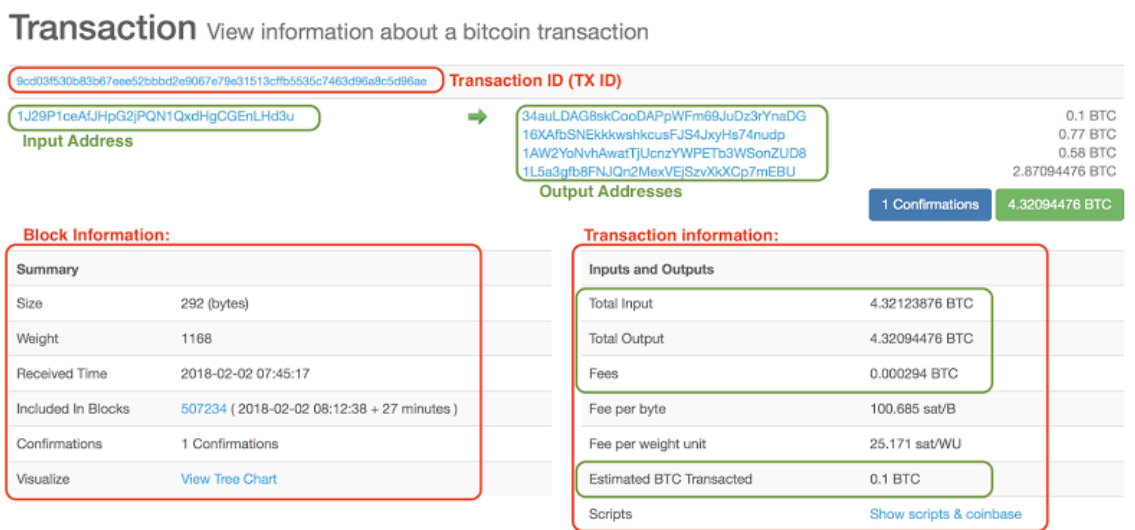
Figure 1: Transaction
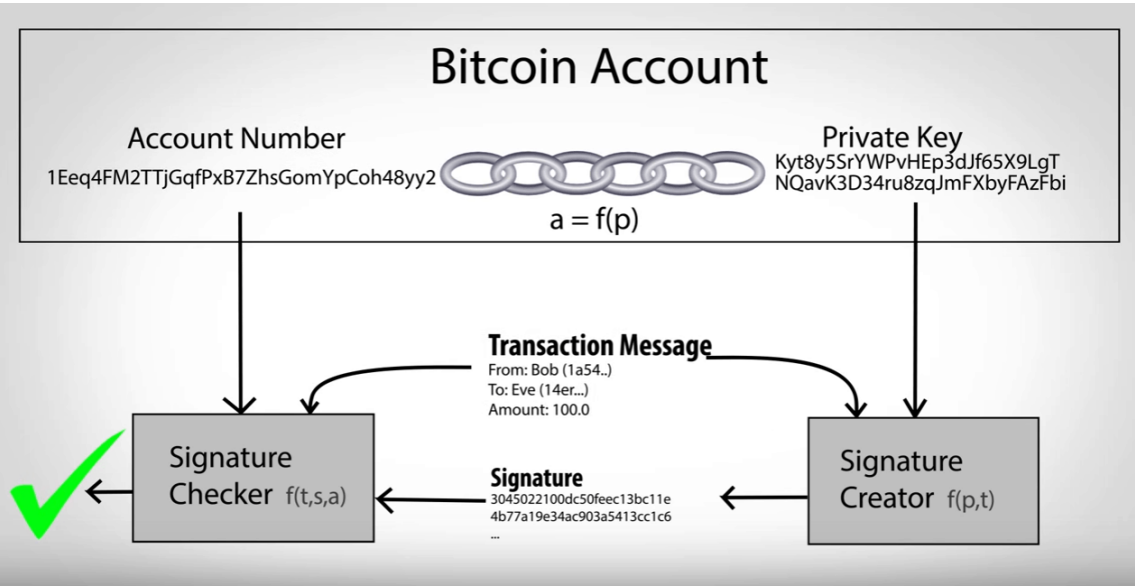
Figure 2: Bitcoin account
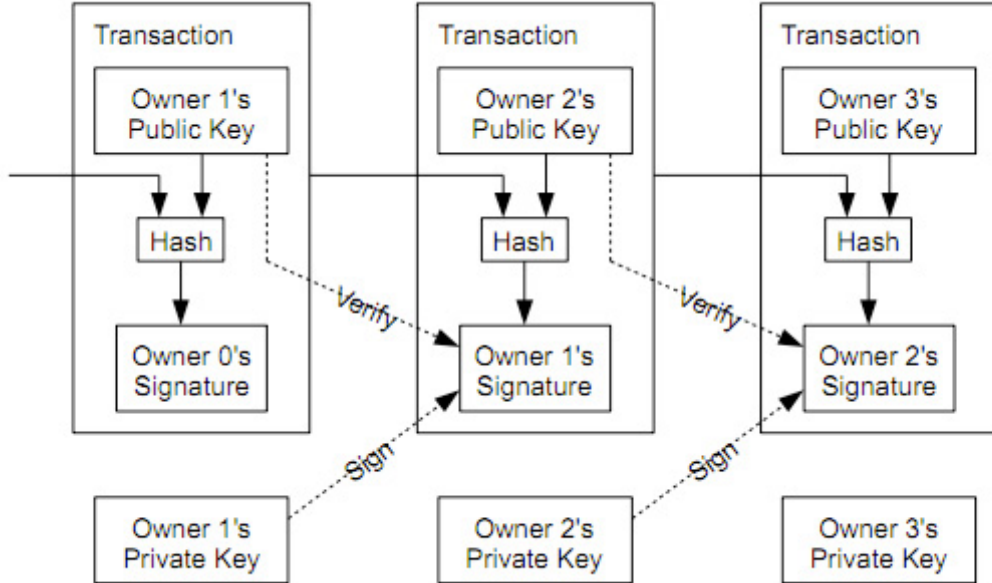
Figure 3: Verification and signature of transactions

In Bitcoin, the miner is someone that calculates a value *nounce* for each block. Because of that, he receives all transaction fees of this block plus a block reward. The mining transaction (made for the miner) does require an input. For each input transaction, it is necessary a signature signed with a private key (like in Figure 2 where the signature is f(p,t)) to prove the ownership of the Bitcoins. With the message and the signature, it is possible to know that the owner of the private key that generates the public key signed the message.

Signature is created from a function that receives as inputs the private key and the message. There is no analytical form of the inverse of this function. Because of that, the only way of finding an element x that this function f so that $f(x) = y$ for a given y is trying for random different values of x. So it is almost impossible to find the private key from a message, a signature, and the public key. In Figure 2, the checker is a f(t,s,a). So because of that, the owner of the private key can sign several messages without anyone knows his private key.

Transactions (shown in Figure 3) are grouped in a block (shown in Figure 4). Each block contains in its header the timestamp of its creation, the hash of the block, the previous hash and a nonce. A nonce is an arbitrary value that the miner has to choose to make the hash of the block respect some specific characteristics.

Each block has a size limit of 1 MB. Because of that, Bitcoin forms a blockchain (a
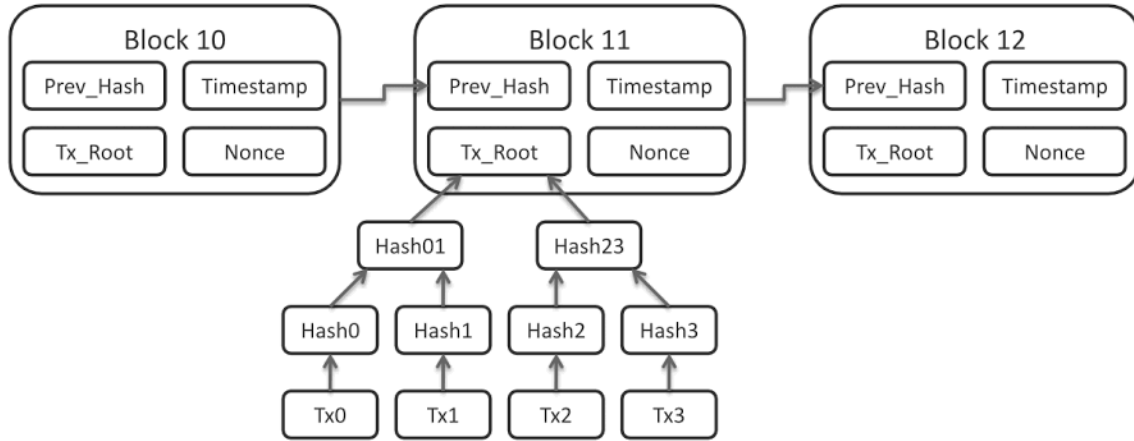
Figure 4: Blockchain

chain of blocks). Each block should be created at an average of 10 minutes. The advantage of using a short average time of block creation is that it will propagate transactions faster in the blockchain. But the disadvantage is that it increases the probability of creating orphan blocks. To make the blockchain tamper-proof (nobody can alter an existing block), there is a concept called proof of work in Bitcoin. To create a new block, the creator (called the miner) has to choose a random value as nonce that makes the hash of the block less than a certain value. This value is chosen in a way that each block should be generated on 10 minutes on average. If the value is for example 100 and the total possibilities of results are 10.000 (codomain), the probability of finding x so that $hash(x) \leqslant 100$ is 1% 100/10.000. Because of that, in this example, the miner has to calculate on average 100 hashes before finding the right one.

If the value is too low, miners will take more time to find a nonce that makes the hash block less than it. If it is too high, it will be easier to find a nonce and they will find it faster.

When two different blocks are created in nearly the same time, there are two valid blockchains. It is because the last block in both blockchains are valid but different. Because of this problem, in the Bitcoin protocol, the largest chain is always the right chain. While two valid chains have the same size, it is not possible to know which chain is the right. This situation is called fork and when it happens, it is necessary to wait to see in which chain the new block will be.

If miners have more than 50% of the hash rates of Bitcoin, they can do an attack called a 51% attack. It happens when some miner, with more power than all network, mine secretly the blocks. So if the main network has 50 blocks, the miner could

produce hidden blocks from 46 to 55 and he would have 10 hidden blocks from the network. When he shows their hidden blocks, his chain becomes the valid chain, because it is bigger. So all transactions from the previous blockchain from 46 to 50 blocks become invalid. Because of that, when someone makes a big transaction in the blockchain, it is a good idea to wait more time. So it is becoming harder and harder to make a 51% with more time. Bitcoin has the highest market value nowadays, so attacking the Bitcoin network is very expensive. Nowadays, this kind of attack is more common in new altcoins.

Ledger is a software that tracks how much money each address can spend. It can be known when calculating the sum amount of all unspent output transactions destined for each address.

The UTXO model used in Bitcoin and the account model used in Ethereum are the two most used kinds of data structures to model account records and savings states.

In the account model, it is saved the address and the balance of each address. For example, the data structure will look like this [(0xabc01, 1.01), (0xabc02, 2.02)]. So the address 0xabc01 has 1.01a of balance and the address 0xabc02 has 2.02 of balance. In this way, it is possible to easily know how much balance each address has, but it is not possible to know how they got in this state.

In the UTXO model (shown in Figure 5), each transaction is saved in the transaction tree. Every transaction is composed of multiples inputs and multiples outputs. But all inputs have to come from outputs that have never been spent before.

Because of that, in the UTXO model, it is easy to make a new transaction from the previous one, but it is harder to know how much each one has. To know how much someone has, it is necessary to calculate the sum of all unspent transaction outputs destined for his addresses.

In the account model, there could be one kind of vulnerability that is less probable to happen in the UTXO model. Changing both addresses balances is not an atomic operation. Because there are two instructions, subtracting the balance of the sender and incrementing the balance of the receiver. If something bad happens between these two instructions, the final state will become an invalid state.

For example:
bobBalance -= 1
Intermediary State
aliceBalance += 1

In the account model, it is straight forward to know how much balance each address has. In the UTXO model, this calculation is made off-chain. It can be a good thing
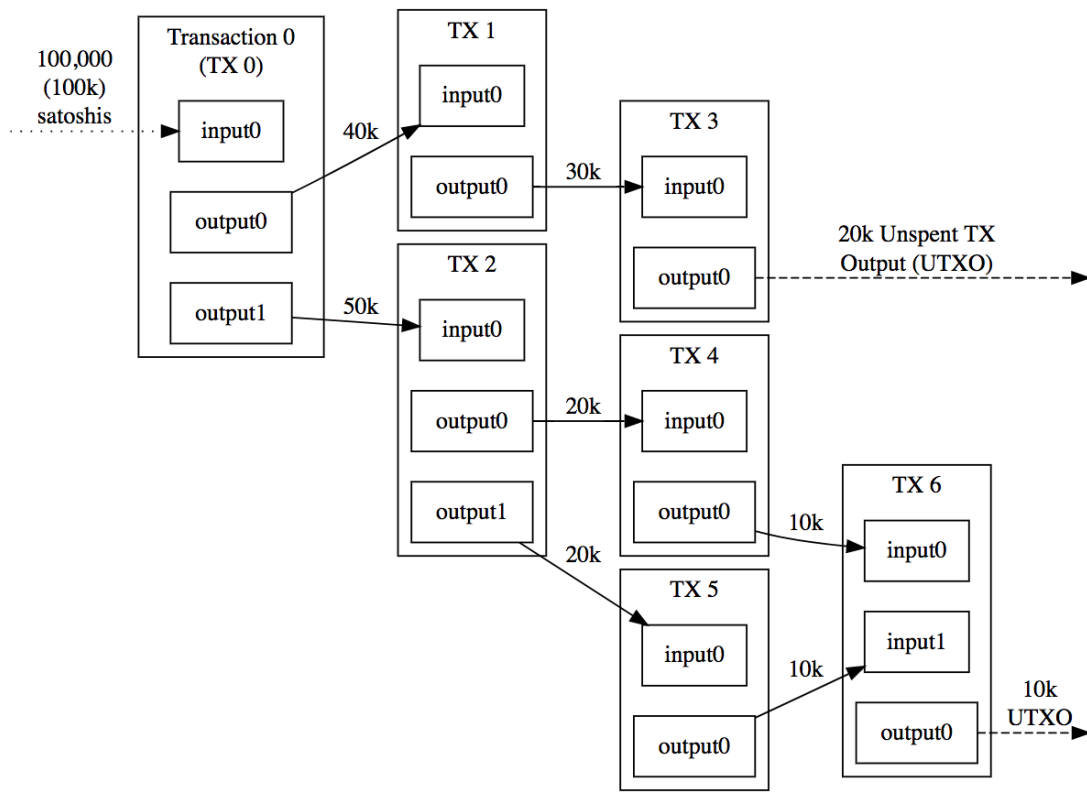
Figure 5: UTXO transactions

because each user has more privacy.

2.3   Ethereum

Ethereum differs from Bitcoin in having an EVM to run script code. EVM is a stack machine and Turing complete (Turing, 1936) while Bitcoin Script is not (it is impossible to do loops and recursion in Bitcoin).

Transactions in Bitcoin are all stored in the blockchain. In Ethereum, just the hash of it is stored in it. So it is saved in the off-chain database. Because of that, it is possible to save more information in Ethereum Blockchain.

Contracts in the blockchain are the script program that comes with the transaction. In Bitcoin, the creator of the contract has to pay the amount proportional to its size. In Ethereum, it is different. Each smart contract in Ethereum is made by a series of instructions. Each instruction consumes different the computational effort. Because of that, in Ethereum, there is a concept of gas, that measure how much computational effort each instruction needs. So in each smart contract, it is well know how much computational effort will be necessary to run it and it is measured in gas. Because computational effort of each node in the Ethereum network is a scarce resource, to execute the smart contract, it is necessary to pay an amount in ether for each gas to the miner run it. Smart contracts that pay more ether per gas run first because the miner will want to have the best profit and they will pick them. If the amount of ether per gas paid is not high enough, the contract will not be executed, because some other contracts pay more that will be executed instead of this one.

Because Ethereum has its EVM with more instructions than Bitcoin and it is Turing Complete, it is considered less secure. Ethereum has its high-level programming language called Solidity that looks like JavaScript.

3 Development

3.1 Crypto Functions

The first thing that we define is the crypto functions that will be needed to make the cryptocurrency. Messages can be defined in multiple ways, one array of bytes, one string or a natural number. Messages in this context means some data.

A private key is a number, a secret that someone has. In Bitcoin, the private key is a 256-bit number. A private key is used to signed messages.

The public key is generated from a private key. But getting the private key from a public key is difficult. To verify who signed a message with a private key, he has to show the public key.

Hash is an injection function (the probability of two functions having the same hash is very low). The function is used from a big domain to a small domain. For example, a hash of the big file (some GBs) is an integer of just some bytes. It is very useful to prove for example that 2 files are equal. If the hash of two files are equal, the probability of these files being equal is high. It is used in torrents clients, so it is safe to download a program to untrusted peers, just have to verify if the hash of the file is equal to the hash of the file wanted.

These functions can be defined, but it is not the purpose of this thesis. So they will be just postulates.

```
postulate _priv≡pub_ : PrivateKey → PublicKey → Set
postulate publicKey2Address : PublicKey → Address
postulate Signed : Msg → PublicKey → Signature → Set
postulate Signed? : (msg : Msg) (pk : PublicKey)
            (sig : Signature) → Dec $ Signed msg pk sig
postulate hashMsg : Msg → Hashed
postulate hash-inj : ∀ m n → hashMsg m ≡ hashMsg n
            → m ≡ n


record SignedWithSigPbk (msg : Msg)(address : Address)
  : Set where
  field
    publicKey  : PublicKey
    pbkCorrect : publicKey2Address publicKey ≡ address
    signature  : Signature
    signed     : Signed msg publicKey signature
```

3.2   Transactions

3.2.1   Definitions

In Bitcoin, there are some transactions. In each transaction, there are multiple inputs and outputs. Each input is named TXFieldWithId. The input of one transaction is the output of another transaction. Firsts outputs are generated from coinbase transaction (created by the miner). Each block has just one of this transaction.

```
data VectorOutput : (time : Time) (size : Nat) (amount : Amount) → Set where
  fstEl : ∀ {time : Time}
    (tx : TXFieldWithId)
    (sameId : TXFieldWithId.time tx ≡ time)
    (elStart : TXFieldWithId.position tx ≡ zero)
    → VectorOutput time 1 (TXFieldWithId.amount tx)

  cons : ∀ {time : Time} {size : Nat} {amount : Amount}
    (listOutput : VectorOutput time size amount)
    (tx : TXFieldWithId)
    (sameId : TXFieldWithId.time tx ≡ time)
    (elStart : TXFieldWithId.position tx ≡ size)
    → VectorOutput time (suc size) (amount + TXFieldWithId.amount tx)
```

Vector output is the vector of outputs transactions. It is a non-empty vector, because it already starts with one element *fstStart* or it is an union from one transaction with another vector. In its representation, it is possible to know in what time it was created (time is the position of they in all transactions), what is his size (quantity of outputs fields) and the total amount spent in this transaction,

elStart is a proof that the position of TXFieldWithId is the last one, because its position in the vector is the same as the last position (size) of the vector. It is used after to specify which input is in the transaction.

```
record TXSigned
  {time    : Time}
  {outSize : Nat}
  {outAmount : Amount}
  (inputs   : List TXFieldWithId)
```

```
(outputs : VectorOutput time outSize outAmount) : Set where
constructor txsig
field
  nonEmpty : NonNil inputs
  signed : All
    (λ input →
    SignedWithSigPbk (txEls→MsgVecOut input outputs)
                     (TXFieldWithId.address input))
                     inputs
  in≥out : txFieldList→TotalAmount inputs ≥ outAmount
```

A signed transaction is composed of a non-empty list of inputs and outputs. For each
input, there is a signature that confirms that he accepted every output in the list of
outputs. And in the transaction, there is proof $(in \geq out)$ that the total amount of
money in all inputs is bigger than the total amount of outputs. The remainder will
be used by the miner.

3.2.2 Raw Transaction

Raw transactions are transactions without any explicit dependent type. Here the
definition of *raw signed transaction*:

```
record RawTXSigned : Set where
  field
    inputs  : List TXFieldWithId
    outputs : List TXFieldWithId
    txSig   : TXSignedRawOutput inputs outputs
```

Raw signed transactions are a record with *inputs*, *outputs* and the signature of *inputs*
and *outputs*.

The definition of some important types:

```
data PublicKey : Set where
  nat : Nat → PublicKey

data Address : Set where
  nat : Nat → Address

data PrivateKey : Set where
```

```
nat : Nat → PrivateKey

data Signature : Set where
  nat : Nat → Signature

data Hashed : Set where
  nat : Nat → Hashed

data Msg : Set where
  nat : (n : Nat) → Msg
  _+msg_ : (m n : Msg) → Msg
```

The definition of *Raw Input*:

```
record RawInput : Set where
  field
    time      : Time
    position  : Nat
    amount    : Amount
    msg       : Msg
    signature : Signature
    publicKey : PublicKey
```

In each input, it is necessary to know the time, the position of it in the transaction, the amount spent, its message, the signature, and its public key. The signature is the signature of the message. And the message is usually related to the amount spent in each output.

The definition of *raw transaction*:

```
record RawTransaction : Set where
  field
    inputs  : List RawInput
    outputs : List TXField
```

It is all inputs and all outputs.

The definition of *Raw TX*:

```
data RawTX : Set where
  coinbase : (tx : RawTXCoinbase) → RawTX
  normalTX : (tx : RawTransaction) → RawTX
```

The definition of *raw transaction coinbase*:

```
record RawTXCoinbase : Set where
  field
    outputs : List TXFieldWithId
```

The definition of *raw Vector Output*:

```
record RawVecOutput (outputs : List TXFieldWithId) : Set where
  field
    time    : Time
    outSize : Nat
    amount  : Amount
    vecOut  : VectorOutput time outSize amount
    proof   : VectorOutput→List vecOut ≡ outputs
```

It has the time, its size, the total amount, the *vector output* and proof that this vector is the same as the list of outputs of this type.

The definition of the record that every input transaction is signed in a given time:

```
record TXSigAll (time : Time) (allInputs : List TXFieldWithId) : Set where
  field
    outSize : Nat
    sub     : SubList allInputs
    amount  : Amount
    outputs : VectorOutput time outSize amount
    signed  : TXSigned (sub→list sub) outputs
```

It has the size of vector output, the sublist of all inputs, the total amount, the *vector output* and a proof that all sublists of inputs are signed.

$rawTXSigned \rightarrow TXSigAll$ returns a signed transaction of all inputs in a given time if *rawTXSigned* has valid signatures for all these inputs:

```
rawTXSigned→TXSigAll : (time : Time) (allInputs : List TXFieldWithId)
  (rawTXSigned : RawTXSigned) → Maybe $ TXSigAll time allInputs
rawTXSigned→TXSigAll time allInputs
  record { outputs = outputs ; txSig = txSig }
  with listTXField→VecOut outputs
... | nothing = nothing
... | just record { outSize = outSize ; vecOut = vecOut ;
```

```
      proof = proofVecOut } with list→subProof allInputs (txSigInput txSig)
...      | nothing = nothing
...      | just record { sub = sub ; proof = proofSub }
           with vecOutTime vecOut == time
...        | no _ = nothing
...        | yes refl = just $ record
    { outSize = outSize ; sub = sub ; outputs = vecOut ; signed = txSigRes }
      where
        txSigRes : TXSigned (sub→list sub) vecOut
        txSigRes rewrite proofSub = txAux
          where
            txAux : TXSigned (txSigInput txSig) vecOut
            txAux rewrite proofVecOut = TXRaw→TXSig vecOut proofVecOut txSig
```

It has to validate first that the *list of outputs* is a valid *Vector Output*. Second, it validates if the signature of the inputs are valid with the *raw signed transaction*. In the last case, it validates if the time of the *vector output* is equal of the time of this transaction. If all conditions match, it returns a proven signed transaction. If not, it returns nothing.

This function transforms a *raw transaction* into a *signed transaction*:

```
TXRaw→TXSig : {inputs : List TXFieldWithId}
  {outputs : List TXFieldWithId}
  {time    : Time}
  {outSize  : Nat}
  {outAmount : Amount}
  (vecOut   : VectorOutput time outSize outAmount)
  (out≡vec : VectorOutput→List vecOut ≡ outputs)
  (txSig    : TXSignedRawOutput inputs outputs)
  → TXSigned inputs vecOut
TXRaw→TXSig {inputs} {outputs} {_} {_} {outAmount} vecOut out≡vec
  record { nonEmpty = (nonEmptyInp , nonNilOutputs) ;
    signed = signed ; in≥out = in≥out } =
  record { nonEmpty = nonEmptyInp ;
    signed = allSigned signed ; in≥out = in≥outProof }
  where
    vecOut≡ListAmount :
      {outAmount : Amount}
      {time    : Time}
      {outSize  : Nat}
```

```
    (outputs : List TXFieldWithId)
    (vecOut   : VectorOutput time outSize outAmount)
    (out≡vec : VectorOutput→List vecOut ≡ outputs)
    → outAmount ≡ txFieldList→TotalAmount outputs
vecOut≡ListAmount [] (fstEl tx sameId elStart) ()
vecOut≡ListAmount [] (cons vecOut tx sameId elStart) ()
vecOut≡ListAmount _
    (fstEl record { time = time ; position = position ; amount = zero ;
    address = address } sameId elStart) refl = refl
vecOut≡ListAmount _ (fstEl record { time = time ;
    position = position ; amount = (suc amount) ;
    address = address } sameId elStart) refl = refl
vecOut≡ListAmount _ (cons vecOut tx sameId elStart) refl =
    let vecProof = vecOut≡ListAmount (VectorOutput→List vecOut) vecOut refl
    in cong (λ x → x + TXFieldWithId.amount tx) vecProof

in≥outProof : txFieldList→TotalAmount inputs ≥ outAmount
in≥outProof rewrite vecOut≡ListAmount outputs vecOut out≡vec = in≥out

sameMessage :
    {outAmount : Amount}
    {time      : Time}
    {outSize   : Nat}
    (outputs   : List TXFieldWithId)
    (input     : TXFieldWithId)
    (nonNilOut : NonNil outputs)
    (vecOut    : VectorOutput time outSize outAmount)
    (out≡vec : VectorOutput→List vecOut ≡ outputs)
    → txEls→Msg input outputs (nonEmptyInp , nonNilOut) ≡
        txEls→MsgVecOut input vecOut
sameMessage _ _ outNotNil (fstEl tx sameId elStart) refl = refl
sameMessage _ _ outNotNil (cons (fstEl tx₁ sameId₁ elStart₁)
    tx sameId elStart) refl = refl
sameMessage _ input unit (cons (cons vecOut tx₂ sameId₂ elStart₂)
    tx₁ sameId₁ elStart₁) refl =
    let msgRest = sameMessage _ input unit (cons vecOut tx₂ sameId₂ elStart₂) refl
    in cong (λ x → TX→Msg (removeId tx₁) +msg x) msgRest

sigPub : {input : TXFieldWithId}
    (sign : SignedWithSigPbk
```

```
        (txEls→Msg input outputs (nonEmptyInp , nonNilOutputs))
        (TXFieldWithId.address input))
    → SignedWithSigPbk (txEls→MsgVecOut input vecOut)
      (TXFieldWithId.address input)
  sigPub {input} sign =
    let msgEq = sameMessage outputs input nonNilOutputs vecOut out≡vec
    in transport (λ msg → SignedWithSigPbk msg
      (TXFieldWithId.address input)) msgEq sign


  allSigned : {inputs : List TXFieldWithId}
    (allSig : All
        (λ input →
          SignedWithSigPbk
            (txEls→Msg input outputs (nonEmptyInp , nonNilOutputs))
            (TXFieldWithId.address input)) inputs)
      → All
        (λ input →
          SignedWithSigPbk (txEls→MsgVecOut input vecOut)
          (TXFieldWithId.address input))
          inputs
  allSigned {[]} allSig = []
  allSigned {input :: inputs} (sig :: allSig) = (sigPub sig) :: (allSigned allSig)
```

The function $vecOut \equiv ListAmount$ returns a proof that the *vector output* is equal to the total amount of the *list of transactions*. It is impossible that the *vector output* is equal to an empty list. In case that the list has just one element, it just has to return *refl*. The another case, it is done recursively.

The proof that the amount of input transaction is greater than the amount of output is just a rewrite from the previous proof ($vecOut \equiv ListAmount outputs vecOut out \equiv vec$).

The function *sameMessage* returns a proof that the message of *raw transaction* is the same as the message of the *vector output*. In case that *vector output* has just size one or two, it is a trivial case. The other cases are doing it recursively.

sigPub is another function that returns a proof that an input message is signed. It validates it with its public key.

The last function returns a proof that every input was signed. It is done in a recursive way using the function *sigPub*.

This is the function that transforms a list of transactions into a possible *vector*

output:

```
listTXField→VecOut : (txs : List TXFieldWithId) → Maybe $ RawVecOutput txs
listTXField→VecOut [] = nothing
listTXField→VecOut (tx :: txs) with listTXField→VecOut txs
... | just vouts = addElementRawVec tx txs vouts
  where
    addElementInVectorOut : {time : Time} {outSize : Nat} {amount : Amount}
      (tx : TXFieldWithId)
      (vecOut : VectorOutput time outSize amount)
      → Maybe $ VectorOutput time (suc outSize)
          (amount + TXFieldWithId.amount tx)
    addElementInVectorOut {time} {outSize} tx vecOut
      with TXFieldWithId.time tx == time
    ... | no ¬p   = nothing
    ... | yes refl with TXFieldWithId.position tx == outSize
    ... | no ¬p = nothing
    ... | yes refl = just $ cons vecOut tx refl refl

    addElementRawVec : (tx : TXFieldWithId)
      (outs : List TXFieldWithId) (vecOut : RawVecOutput outs)
      → Maybe $ RawVecOutput (tx :: outs)
    addElementRawVec tx outs record { time = time ; outSize = outSize ;
                                      vecOut = vecOut ; proof = proof }
      with addElementInVectorOut tx vecOut
    ... | nothing = nothing
    ... | just vec with TXFieldWithId.time tx == time
    ... | no _ = nothing
    ... | yes refl with TXFieldWithId.position tx == outSize
    ...      | no _ = nothing
    ...      | yes refl = just $ record { time = time ; outSize = suc outSize
      ; vecOut = cons vecOut tx refl refl ; proof = cong (_::_ tx) proof }
... | nothing with txs == []
...      | no _ = nothing
...      | yes p rewrite p = createVecOutsize tx
```

The list has to be at least with a size one. Because the *vector output* can not be empty. To add one element into the vector, it has to verify if the time is equal to the first time. Another verification is that the informed position in the vector is right. If all validations are right, it returns the vector output. If it is not, it returns nothing.

The definition of the function that transform a *raw transaction* into a *raw signed transaction*:

```
raw→TXSigned : ∀ (time : Time) (ftx : RawTransaction)
  → Maybe RawTXSigned
raw→TXSigned time record { inputs = inputs ; outputs = outputs }
  with NonNil? inputs
... | no _ = nothing
... | yes nonNilInp with NonNil? outputs
... | no _ = nothing
... | yes nonNilOut = ans
  where
    inpsField : List TXFieldWithId
    inpsField = map raw→TXField inputs

    outsField : List TXFieldWithId
    outsField = addId zero time outputs

    nonNilMap : ∀ {A B : Set} {f : A → B} (lista : List A)
      → NonNil lista → NonNil (map f lista)
    nonNilMap [] ()
    nonNilMap (_ :: _) nla = unit

    nonNilImpTX : NonNil inpsField
    nonNilImpTX = nonNilMap inputs nonNilInp

    nonNilAddId : {time : Time} (outputs : List TXField)
      (nonNilOut : NonNil outputs)
      → NonNil (addId zero time outputs)
    nonNilAddId [] ()
    nonNilAddId (_ :: outputs) nonNil = nonNil

    nonNilOutTX : NonNil outsField
    nonNilOutTX = nonNilAddId outputs nonNilOut

    nonEmpty : NonNil inpsField × NonNil outsField
    nonEmpty = nonNilImpTX , nonNilOutTX

    All?Signed : (inputs : List RawInput) →
      Maybe (All (λ input → SignedWithSigPbk
      (txEls→Msg input outsField nonEmpty)
```

```
            (TXFieldWithId.address input)) (map raw→TXField inputs))
      All?Signed [] = just []
      All?Signed (input :: inputs)
        with Signed? (txEls→Msg (raw→TXField input) outsField nonEmpty)
        (RawInput.publicKey input) (RawInput.signature input)
      ... | no _ = nothing
      ... | yes signed with All?Signed inputs
      ... | nothing = nothing
      ... | just allInputs = just $ (record
                                    { publicKey = RawInput.publicKey input
                                    ; pbkCorrect = refl
                                    ; signature = RawInput.signature input
                                    ; signed = signed
                                    }) :: allInputs


      in≥out : Dec $ txFieldList→TotalAmount inpsField ≥
                     txFieldList→TotalAmount outsField
      in≥out = txFieldList→TotalAmount inpsField ≥?p
               txFieldList→TotalAmount outsField


      ans : Maybe RawTXSigned
      ans with All?Signed inputs
      ... | nothing = nothing
      ... | just signed with in≥out
      ... | no _ = nothing
      ... | yes in>out = just $ record { inputs = inpsField ; outputs = outsField ;
          txSig = record { nonEmpty = nonEmpty ; signed = signed ; in≥out = in>out } }
```

The first validation that the function does is verifying that the outputs are not empty. Another validation is verifying if the amount spent on inputs is greater than the amount of the outputs. The function *Signed?*, defined in the crypto library, validates if the message was signed with the input. After, it validates if all inputs are signed. If all validations are right, it returns the *raw transaction signed*. If it is not, it returns nothing.

3.3   Transaction Tree

3.3.1   Definition

The transaction tree is one of the most important data structures in Bitcoin. In every new transaction, the UTXOs used as input is removed from the transaction tree.

```
mutual
  data TXTree : (time : Time) (block : Nat)
    (outputs : List TXFieldWithId)
    (totalFees : Amount)
    (qtTransactions : tQtTxs) → Set where

    genesisTree : TXTree (nat zero) zero [] zero zero
    txtree :
      {block : Nat} {time : Time}
      {outSize : Nat} {amount : Amount}
      {inputs : List TXFieldWithId}
      {outputTX : VectorOutput time outSize amount}
      {totalFees : Amount} {qtTransactions : tQtTxs}
      (tree : TXTree time block inputs totalFees qtTransactions)
      (tx : TX {time} {block} {inputs} {outSize} tree outputTX)
      (proofLessQtTX :
        Either
          (IsTrue (lessNat (finToNat qtTransactions) totalQtSub1))
          (isCoinbase tx))
    → TXTree (sucTime time)
        (nextBlock tx)
        (inputsTX tx ++ VectorOutput→List outputTX)
        (incFees tx) (incQtTx tx proofLessQtTX)
```

In this implementation, time is the number of transactions in TXTree. Block is related to which block the transaction tree is. After every new coinbase transaction (the miner transaction), the block size (*blocksize*) increment in one quantity. Total fees (*totalFees*) are how much the miner will have in fee of transactions if he makes a block with these transactions. Quantity of transactions (*qtTransaction*) is how many transactions there are in the current block. The type is tQtTxs instead of a natural number because, in this implementation, each block can have a maximum number of transactions. In Bitcoin, it is different, each block has a limit size in space of 1 MB.

Genesis tree is the first case. It is when the cryptocurrency was created. *txtree* is created from another tree. *proofLessQtTX* is a proof that the last transaction tree has its block size less than the maximum block size minus one or it is a coinbase transaction. It is because it is necessary to verify the size of the last *txtree* so it will not have the size greater than the maximum.

```
data TX {time : Time} {block : Nat} {inputs : List TXFieldWithId}
      {outSize : Nat} {outAmount : Amount}
      {totalFees : Nat} {qtTransactions : tQtTxs}
    : (tr : TXTree time block inputs totalFees qtTransactions)
      (outputs : VectorOutput time outSize outAmount) → Set where
  normalTX :
      (tr : TXTree time block inputs totalFees qtTransactions)
      (SubInputs : SubList inputs)
      (outputs : VectorOutput time outSize outAmount)
      (txSigned : TXSigned (sub→list SubInputs) outputs)
      → TX tr outputs
  coinbase :
      (tr : TXTree time block inputs totalFees qtTransactions)
      (outputs : VectorOutput time outSize outAmount)
      (pAmountFee : outAmount out≡Fee totalFees +RewardBlock block)
      → TX tr outputs
```

TX is related to the transaction done in the cryptocurrency. There are two kinds of transactions. Coinbase transaction is the transaction done by the miner. In coinbase, they have just outputs and do not have any input. *pAmountFee* is proof that the output of the coinbase transaction is equal to the total fees plus a block reward.

Another kind of transaction is the *normalTX*, a regular transaction. *SubInputs* are a sub-list of all unspent transaction outputs of the previous transaction tree. Outputs are the new unspent transaction from this transaction. So who receives the amount from this transaction can spend it after. *TxSigned* is the signature that proves that every owner of each input approve this transaction. In *TxSigned*, there is proof that the output amount is greater than the input amount too.

```
isCoinbase : ∀ {block : Nat} {time : Time}
    {inputs : List TXFieldWithId}
    {outSize : Nat} {amount : Amount}
    {totalFees : Nat} {qtTransactions : tQtTxs}
    {tr : TXTree time block inputs totalFees qtTransactions}
```

```
  {outputs : VectorOutput time outSize amount}
  (tx : TX {time} {block} {inputs} {outSize} tr outputs)
  → Set
isCoinbase (normalTX _ _ _ _) = ⊥
isCoinbase (coinbase _ _ _) = ⊤
```

This function just returns trivial type if coinbase and bot type if not.

```
nextBlock : ∀ {block : Nat} {time : Time}
  {inputs : List TXFieldWithId}
  {outSize : Nat} {amount : Amount}
  {totalFees : Nat} {qtTransactions : tQtTxs}
  {tr : TXTree time block inputs totalFees qtTransactions}
  {outputs : VectorOutput time outSize amount}
  (tx : TX {time} {block} {inputs} {outSize} tr outputs)
  → Nat
nextBlock (normalTX genesisTree _ _ _) = zero
nextBlock {block} (normalTX (txtree _ (normalTX _ _ _ _) _) _ _ _) = block
nextBlock {block} (normalTX (txtree _ (coinbase _ _ _) _) _ _ _) = suc block
nextBlock (coinbase genesisTree _ _) = zero
nextBlock {block} (coinbase (txtree _ (normalTX _ _ _ _) _) _ _) = block
nextBlock {block} (coinbase (txtree _ (coinbase _ _ _) _) _ _) = suc block
```

If it is a normal transaction, the block continues the same. If it is a coinbase transaction, the next transaction tree will be in a new block.

```
incQtTx : ∀ {qtTransactions : tQtTxs}
  {block : Nat} {time : Time}
  {inputs : List TXFieldWithId}
  {outSize : Nat} {amount : Amount}
  {totalFees : Nat}
  {tr : TXTree time block inputs totalFees qtTransactions}
  {outputs : VectorOutput time outSize amount}
  (tx : TX {time} {block} {inputs} {outSize} tr outputs)
  (proofLessQtTX :
    Either
      (IsTrue (lessNat (finToNat qtTransactions) totalQtSub1))
      (isCoinbase tx))
  → tQtTxs
incQtTx {qt} (normalTX _ _ _ _) (left pLess) =
  natToFin (suc (finToNat qt)) {{pLess}}
```

```
incQtTx {qt} (normalTX _ _ _ _) (right ())
incQtTx (coinbase _ _ _) _ = zero
```

This function is to increment the number of transactions in the block. It has to receive proof that the quantity of transaction that was before this new transaction was less than then the maximum quantity of transactions allowed. So it is guaranteed that the number of transactions will never be greater than the maximum allowed. If it is a coinbase transaction, it will be a new block. So the number of transactions starts being zero.

```
incFees : ∀ {block : Nat} {time : Time}
  {inputs : List TXFieldWithId}
  {outSize : Nat} {amount : Amount}
  {totalFees : Amount} {qtTransactions : tQtTxs}
  {tr : TXTree time block inputs totalFees qtTransactions}
  {outputs : VectorOutput time outSize amount}
  (tx : TX {time} {block} {inputs} {outSize} tr outputs)
  → Amount
incFees {_} {_} {_} {_} {amount} {totalFees}
  (normalTX _ SubInputs _ (txsig _ _ in≥out)) =
  txFieldList→TotalAmount (sub→list SubInputs)
  - amount p≥ in≥out
  + totalFees
incFees (coinbase tr outputs _) = zero
```

IncFees is a function that increments how much fee the miner will receive. If it is a coinbase transaction, the fee will be received by the miner, so the next miner will not receive this previous fee. Because of that, the new fee will start from zero. If it is a normal transaction, the newest fee will be the amount of input of the transaction minus the output of this transaction plus the last fee of previous transactions.

```
_out≡Fee_+RewardBlock_ : (amount : Amount)
  (totalFees : Amount)
  (block : Nat) → Set
amount out≡Fee totalFees +RewardBlock block =
  amount ≡ totalFees + blockReward block
```

$\_out \equiv Fee\_ + RewardBlock\_$ is a proof that the amount of output transactions is equal to total fees of other transactions plus the block reward.

3.3.2   Raw Transaction Tree

The raw transaction tree is the tree without the explicit types. Here, the definition:

```
record RawTXTree : Set where
  field
    time       : Time
    block      : Nat
    outputs    : List TXFieldWithId
    totalFees  : Amount
    qtTransactions : tQtTxs
    txTree     : TXTree time block outputs totalFees qtTransactions
```

A good advantage of raw data types is that they do not use dependent types. They are made from primitive types like lists, integers, boolean, strings. So it become easier to serialize and deserialize them. *addTransactionTree* is a function that adds a transaction to a transaction tree. If this transaction is compatible with the transaction tree, it returns a new transaction tree. If it is not compatible, it returns nothing. A better solution is a proof that this transaction is invalid with the transaction tree instead of nothing. But defining what is an invalid transaction can be tricky, because it has to derive absurd from a valid transaction and all other inputs that made this transaction.

```
addTransactionTree : (txTree : RawTXTree) (tx : RawTX) → Maybe RawTXTree
addTransactionTree record { time = time ; block = block ; outputs = outputs ;
  qtTransactions = qtTransactions ; totalFees = totalFees ; txTree = txTree }
  (coinbase record { outputs = outputsTX }) with listTXField→VecOut outputsTX
... | nothing = nothing
... | just record { time = timeOut ; outSize = outSize ; vecOut = vecOut }
  with vecOut→Amount vecOut == totalFees + blockReward block
... | no _ = nothing
... | yes eqBlockReward
  with time == timeOut
... | no _ = nothing
... | yes refl = just $
  record { time = sucTime time ;
  block = nextBlock (coinbase txTree vecOut eqBlockReward) ;
  outputs = outputs ++ VectorOutput→List vecOut ;
  txTree = txtree txTree tx (right unit) }
  where
    tx : TX txTree vecOut
    tx = coinbase txTree vecOut eqBlockReward
```

There are two types of transactions. The first one is when the transaction is a coinbase transaction. The function tries first to transform a list of *TXField* into *VecOut*. If it can not transforms, it returns nothing. If it can, it validates if the amount of vector output is equal to total fees plus the block reward. After, it validates if the time of the transaction is equal to the time of the transaction tree. In the end, it adds the outputs of the transaction to the vector of outputs. Because it is a coinbase transaction, there are no inputs to be removed.

```
addTransactionTree record { time = time ; block = block ; outputs = outputs ;
    qtTransactions = qtTransactions ; txTree = txTree }
    (normalTX record { inputs = inputsTX ; outputs = outputsTX })
    with dec< (finToNat qtTransactions) totalQtSub1
... | no _ = nothing
... | yes pLess
    with raw→TXSigned time record { inputs = inputsTX ; outputs = outputsTX }
... | nothing = nothing
... | just txSig with rawTXSigned→TXSigAll time outputs txSig
... | nothing = nothing
... | just record { outSize = outSize ; sub = sub ;
          outputs = outs ; signed = signed } =
    just $ record { time = sucTime time ;
          block = nextBlock (normalTX txTree sub outs signed) ;
    outputs = list-sub sub ++ VectorOutput→List outs ;
    txTree = txtree txTree (normalTX txTree sub outs signed) (left pLess) }
```

The second case is when the transaction is regular, looks like the same. First, The function validates if the quantity of transactions is less than the maximum allowed. Second, it validates if this transaction is a valid signed transaction. If all these conditions are true, it returns a new transaction tree with news outputs equal to the outputs of this transaction plus the outputs of the last transaction tree minus the inputs. In case of an invalid transaction, the function returns nothing.

3.3.3 Proofs

One of the important proofs is that each output of the *outputs transaction* is distinct. This is very important because it guarantees that each input in the transaction could be just related to just one unspent output. This characteristic could be in the type of transaction tree, but it is proven outside of it.

First, it is necessary to define what is a distinct union of two distinct lists:

unionDistinct : {A : Set} {la lb : List A} (da : Distinct la) (db : Distinct lb)
  ($twoDist$: twoListDistinct la lb) → Distinct $ la ++ lb
unionDistinct {_} {[]} {lb} da db $twoDist$ = db
unionDistinct {_} {_} {lb} (cons x da $isDistXla$) db ($isDistXlb$, $distLaLb$) =
  cons x (unionDistinct da db $distLaLb$) (isDistUnion x $isDistXla$ $isDistXlb$)

The union of distinct lists makes a new distinct list if both are distinct to each other.

Now, to prove that transactions outputs are a distinct list:

uniqueOutputs : {$time$: Time}
  {$block$: Nat}
  {$outputs$: List TXFieldWithId}
  {$totalFees$: Amount}
  {$qtTransactions$: tQtTxs}
  ($txTree$: TXTree $time$ $block$ $outputs$ $totalFees$ $qtTransactions$)
  → Distinct $outputs$
uniqueOutputs genesisTree = []
uniqueOutputs (txtree {$block$} {$time$} {$outSize$} {$inputs$} {_} {$vecOut$} $tree$ tx _) =
  unionDistinct {_} {inputsTX tx} {VectorOutput→List $vecOut$}
  (distInputs tx) (vecOutDist $vecOut$)
  (allDistincts (inputsTXTimeLess tx) (allVecOutSameTime $vecOut$))

In the first case, the transaction tree is a genesis tree without any outputs. So an empty list is a distinct list. In the second case, the outputs are the union of inputs of the transaction with the outputs of *vector output*. So, it is necessary to prove that inputs of the transaction are distinct, that elements of *vector output* are also distinct and that both lists are distinct to each other.

distInputs : {$time$: Time}
  {$block$: Nat}
  {$inputs$: List TXFieldWithId}
  {$outSize$: Nat}
  {$totalFees$: Amount}
  {$qtTransactions$: tQtTxs}
  {$outAmount$: Amount}
  {$tree$: TXTree $time$ $block$ $inputs$ $totalFees$ $qtTransactions$}
  {$outVec$: VectorOutput $time$ $outSize$ $outAmount$}
  (tx : TX $tree$ $outVec$)
  → Distinct $ inputsTX tx
distInputs (normalTX genesisTree [] $outputs$ $txSigned$) = []

```
distInputs (normalTX (txtree {_} {_} {_} {_} {_} {vecOut} tr tx _)
    SubInputs outputs txSigned) =
    distList→distSub {_} {_} {SubInputs} (unionDistinct {_} {inputsTX tx}
    (distInputs tx) (vecOutDist vecOut)
    (allDistincts (inputsTXTimeLess tx) (allVecOutSameTime vecOut)))
distInputs (coinbase genesisTree outVec _) = []
distInputs (coinbase
    (txtree {_} {_} {_} {_} {_} {vecOut} tr tx _) outVec _) =
    unionDistinct {_} {inputsTX tx} (distInputs tx)
    (vecOutDist vecOut)
    (allDistincts (inputsTXTimeLess tx) (allVecOutSameTime vecOut) )
```

There are some cases to prove that inputs are distinct. First, if it is a regular transaction or if it is a coinbase transaction. Second, if the transaction tree of this transaction is a genesis tree or if it is a regular tree.

If the transaction tree of the transaction is a genesis tree (the first tree), the number of inputs is zero. So they are distinct.

In other cases, it does the same thing as proof of unique outputs. The only difference is that it also does a recursive proof. It assumes that the transaction of the last transaction tree is also distinct.

```
allDistincts : {time : Time} {vec< vec≡ : List TXFieldWithId}
    (all< : All (λ tx → tx out<time time) vec<)
    (all≡ : All (λ tx → TXFieldWithId.time tx ≡ time) vec≡)
    → twoListDistinct vec< vec≡
allDistincts {time} {.[]} {vec≡} [] all≡ = unit
allDistincts {time} {(x :: _)} {vec≡} (p< :: all<) all≡ =
    distinctLess all≡ , allDistincts all< all≡
    where
      sucRemove : ∀ {m n : Nat} (suc≡ : _≡_ {_} {Nat}
        (suc m) (suc n)) → m ≡ n
      sucRemove refl = refl

      ¬n≡suck+n : (k n : Nat) → ¬ (n ≡ suc k + n)
      ¬n≡suck+n k zero ()
      ¬n≡suck+n k (suc n) eqs = ¬n≡suck+n k n let eq = sucRemove eqs in
        trans eq (add-suc-r k n)

      ¬n<n : {n : Nat} → ¬ (n < n)
      ¬n<n {n} (diff k eq) = ¬n≡suck+n k n eq
```

```
distinctLess : {vec≡ : List TXFieldWithId}
  (all≡ : All (λ tx → TXFieldWithId.time tx ≡ time) vec≡)
  → isDistinct x vec≡
distinctLess [] = unit
distinctLess (refl :: all≡) = (λ{ refl → ¬n<n p<}) , (distinctLess all≡)
```

Both are distinct to each other because all of the transactions of input has the timeless then the time of the transaction. And because all of the outputs of the current transaction has time equal to the current time of this transaction.

```
outputsTimeLess :
  {time : Time}
  {block : Nat}
  {outputs : List TXFieldWithId}
  {totalFees : Amount}
  {qtTransactions : tQtTxs}
  (txTree : TXTree time block outputs totalFees qtTransactions )
  → All (λ output → output out<time time) outputs
outputsTimeLess genesisTree = []
outputsTimeLess {_} {_} {_} {totalFees} {qtTransactions}
  (txtree {block} {time} {amount} {outSize} {outputs} {outVec} txTree tx _) =
  allJoin (inputsTX tx) (VectorOutput→List outVec)
  (inputsTree→inputsTXtx tx $ outputsTimeLess txTree)
  $ vecOutTimeLess outVec
  where
    vecOutTimeLess : {time : Time}
      {outSize : Nat}
      {amount : Amount}
      (vecOut : VectorOutput time outSize amount)
      → All (λ output → output out<time (sucTime time))
      (VectorOutput→List vecOut)
    vecOutTimeLess (fstEl tx refl elStart) =
      (diff zero (timeToNatSuc {TXFieldWithId.time tx})) :: []
    vecOutTimeLess (cons {time} vecOut tx refl elStart) =
      (diff zero (timeToNatSuc {time})) :: (vecOutTimeLess vecOut)

    ≤timeSuc : {t1 : TXFieldWithId} {t2 : Time} (pt : t1 out<time t2)
      → t1 out<time (sucTime t2)
    ≤timeSuc {txfieldid time position amount address} {t2}
      (diff k eq) = diff (suc k) (trans (eqTimeNat {t2}) eqsuc)
```

```
   where
     eqsuc : _ ≡ _ {_} {Nat} (suc (timeToNat t2))
       (suc (suc (k + timeToNat time)))
     eqsuc = cong suc eq

     eqTimeNat : {t2 : Time} → timeToNat (sucTime t2) ≡ suc (timeToNat t2)
     eqTimeNat {nat zero} = refl
     eqTimeNat {nat (suc x)} = refl

 inputsTree→inputsTXtx : {inputs : List TXFieldWithId}
   {totalFees : Amount}
   {qtTransactions : Fin totalQt}
   {tree : TXTree time block inputs totalFees qtTransactions}
   (tx : TX tree outVec)
   (allInps : All (λ output → output out<time time) inputs)
   → All (λ input → input out<time sucTime time) (inputsTX tx)
 inputsTree→inputsTXtx {[]} (normalTX tr [] outVec txSigned) [] = []
 inputsTree→inputsTXtx {[]} (coinbase tr outputs _) [] = []
 inputsTree→inputsTXtx {input :: inputs} (normalTX tr (input ¬:: SubInputs)
   outVec txSigned) (pt :: allInps) =
   ≤timeSuc {input} {time} pt :: allProofFG (λ y pf → ≤timeSuc {y} {time} pf)
   (allList→allSub SubInputs allInps)
 inputsTree→inputsTXtx {input :: inputs} (normalTX tr (input :: SubInputs)
   outVec txSigned) (x :: allInps) =
   allProofFG (λ y pf → ≤timeSuc {y} {time} pf)
   (allList→allSub SubInputs allInps)
 inputsTree→inputsTXtx {input :: inputs} (coinbase tr outVec _)
   (pt :: allInps) = ≤timeSuc {input} {time} pt
   :: allProofFG (λ y pf → ≤timeSuc {y} {time} pf) allInps
```

The proof that the time of the outputs is less than the current time of the transaction is done recursively. It is both necessary to proof that *inputs of tx* and *vector output* have both times less than the current time of this transaction. It is all done recursively.

3.4   Ledger

Ledger is necessary for users to send their coins or to know how much money they have in total.

Here, the definition of how much money the user has in the last tree:

```
ledgerTree : (rawTXTree : RawTXTree) (addr : Address) → Amount
ledgerTree txTree = ledgerOut outputs
  where open RawTXTree.RawTXTree txTree
```

The definition of *ledgerOut*:

```
ledgerOut : ∀ (outputs : List TXFieldWithId) (addr : Address)
  → Amount
ledgerOut [] addr = zero
ledgerOut (output :: outputs) addr with TXFieldWithId.address output == addr
... | yes _ = TXFieldWithId.amount output + ledgerOut outputs addr
... | no _ = ledgerOut outputs addr
```

If there is no output, it returns zero of the amount. If there is at least one output, it verifies if the output address is the same as the address. If it is, it adds the amount to the amount of the rest of the outputs. If it is not, it just returns the result of the recursion of the rest of the outputs.

Here, the same code for list of outputs without id:

```
ledgerOutNoId : ∀ (outputs : List TXField) (addr : Address)
  → Amount
ledgerOutNoId [] addr = zero
ledgerOutNoId (output :: outputs) addr with TXField.address output == addr
... | yes _ = TXField.amount output + ledgerOutNoId outputs addr
... | no _ = ledgerOutNoId outputs addr
```

3.5   Blockchain

3.5.1   Definition

This is how a block is defined in this work:

```
record Block
  {block₁ : Nat}
  {time₁ : Time}
  {outputs₁ : List TXFieldWithId}
  {totalFees₁ : Amount}
  {qtTransactions₁ : tQtTxs}
  (txTree₁ : TXTree time₁ block₁ outputs₁ totalFees₁ qtTransactions₁)

  {time₂ : Time}
  {outputs₂ : List TXFieldWithId}
  {totalFees₂ : Amount}
  {qtTransactions₂ : tQtTxs}
  (txTree₂ : TXTree time₂ block₁ outputs₂ totalFees₂ qtTransactions₂)
  : Set where
  constructor blockc
  field
    nxTree    : nextTXTree txTree₁ txTree₂
    fstBlock : firstTreesInBlock txTree₁
    sndBlockCoinbase : coinbaseTree txTree₂
```

nextTXTree assures that the second transaction tree is from the first transaction tree. *firstTreesInBlock* guarantees that the last transaction in the first transaction tree is the first in the block. *coinBaseTree* assures that the last transaction in the second transaction tree is a coinbase transaction.

Blockchain is a chain of valid blocks. Every new block must be a continuation of the previous one. Here is the definition of the blockchain:

```
data Blockchain :
  {block₁ : Nat}
  {time₁ : Time}
  {outputs₁ : List TXFieldWithId}
  {totalFees₁ : Amount}
  {qtTransactions₁ : tQtTxs}
  {txTree₁ : TXTree time₁ block₁ outputs₁ totalFees₁ qtTransactions₁}
```

$\{time_2 : \mathsf{Time}\}$
$\{outputs_2 : \mathsf{List\ TXFieldWithId}\}$
$\{totalFees_2 : \mathsf{Amount}\}$
$\{qtTransactions_2 : \mathsf{tQtTxs}\}$
$\{txTree_2 : \mathsf{TXTree}\ time_2\ block_1\ outputs_2\ totalFees_2\ qtTransactions_2\}$
$(block : \mathsf{Block}\ txTree_1\ txTree_2)$
$\rightarrow \mathsf{Set}\ \mathsf{where}$
   fstBlock :
      $\{block_1 : \mathsf{Nat}\}$
      $\{time_1 : \mathsf{Time}\}$
      $\{outputs_1 : \mathsf{List\ TXFieldWithId}\}$
      $\{totalFees_1 : \mathsf{Amount}\}$
      $\{qtTransactions_1 : \mathsf{tQtTxs}\}$
      $\{txTree_1 : \mathsf{TXTree}\ time_1\ block_1\ outputs_1\ totalFees_1\ qtTransactions_1\}$

      $\{time_2 : \mathsf{Time}\}$
      $\{outputs_2 : \mathsf{List\ TXFieldWithId}\}$
      $\{totalFees_2 : \mathsf{Amount}\}$
      $\{qtTransactions_2 : \mathsf{tQtTxs}\}$
      $\{txTree_2 : \mathsf{TXTree}\ time_2\ block_1\ outputs_2\ totalFees_2\ qtTransactions_2\}$
      $(block : \mathsf{Block}\ txTree_1\ txTree_2)$
      $\rightarrow \mathsf{Blockchain}\ block$

   addBlock :
      $\{block\text{-}p_1 : \mathsf{Nat}\}$
      $\{time\text{-}p_1 : \mathsf{Time}\}$
      $\{outputs\text{-}p_1 : \mathsf{List\ TXFieldWithId}\}$
      $\{totalFees\text{-}p_1 : \mathsf{Amount}\}$
      $\{qtTransactions\text{-}p_1 : \mathsf{tQtTxs}\}$
      $\{txTree\text{-}p_1 : \mathsf{TXTree}\ time\text{-}p_1\ block\text{-}p_1\ outputs\text{-}p_1\ totalFees\text{-}p_1\ qtTransactions\text{-}p_1\}$

      $\{time\text{-}p_2 : \mathsf{Time}\}$
      $\{outputs\text{-}p_2 : \mathsf{List\ TXFieldWithId}\}$
      $\{totalFees\text{-}p_2 : \mathsf{Amount}\}$
      $\{qtTransactions\text{-}p_2 : \mathsf{tQtTxs}\}$
      $\{txTree\text{-}p_2 : \mathsf{TXTree}\ time\text{-}p_2\ block\text{-}p_1\ outputs\text{-}p_2\ totalFees\text{-}p_2\ qtTransactions\text{-}p_2\}$
      $\{block\text{-}p : \mathsf{Block}\ txTree\text{-}p_1\ txTree\text{-}p_2\}$
      $(blockchain : \mathsf{Blockchain}\ block\text{-}p)$

$\{outSize : \mathsf{Nat}\}$
$\{amount : \mathsf{Amount}\}$
$\{outputTX : \mathsf{VectorOutput}\ time\text{-}p_2\ outSize\ amount\}$
$\{tx : \mathsf{TX}\ \{time\text{-}p_2\}\ \{block\text{-}p_1\}\ \{outputs\text{-}p_2\}\ \{outSize\}\ txTree\text{-}p_2\ outputTX\}$
$\{proofLessQtTX :$
  Either
    $(\mathsf{IsTrue}\ (\mathsf{lessNat}\ (\mathsf{finToNat}\ qtTransactions\text{-}p_2)\ \mathsf{totalQtSub1}))$
    $(\mathsf{isCoinbase}\ tx)\}$

$\{time_2 : \mathsf{Time}\}$
$\{outputs_2 : \mathsf{List}\ \mathsf{TXFieldWithId}\}$
$\{totalFees_2 : \mathsf{Amount}\}$
$\{qtTransactions_2 : \mathsf{tQtTxs}\}$
$\{txTree_2 : \mathsf{TXTree}\ time_2\ (\mathsf{nextBlock}\ tx)\ outputs_2\ totalFees_2\ qtTransactions_2\}$
$(block : \mathsf{Block}\ (\mathsf{txtree}\ txTree\text{-}p_2\ tx\ proofLessQtTX)\ txTree_2)$
$\rightarrow \mathsf{Blockchain}\ block$

In the first case, blockchain just has one block, called *fstBlock*. In the second case, the blockchain is an addition of a valid block from a previous blockchain.

3.5.2 Creation

In this section, there will be explanation of how blockchain is created from blocks and how blocks are created from transaction trees. To create a blockchain, it is first needed to create the last block. From the last block, it is possible to create all the chain.

```
block→blockchain : ∀
  {block₁ time₁ outputs₁ totalFees₁ qtTransactions₁}
  {txTree₁ : TXTree time₁ block₁ outputs₁ totalFees₁ qtTransactions₁}
  {time₂ outputs₂ totalFees₂ qtTransactions₂}
  {txTree₂ : TXTree time₂ block₁ outputs₂ totalFees₂ qtTransactions₂}
  (block : Block txTree₁ txTree₂)
  → Blockchain block
block→blockchain {_} {_} {_} {_} {_} {genesisTree}
  (blockc nxTree fstBlock₁ sndBlockCoinbase) =
  fstBlock (blockc nxTree unit sndBlockCoinbase)
block→blockchain {_} {_} {_} {_} {_} {txtree tree tx proofLessQtTX}
  (blockc nxTree fstBlock₁ sndBlockCoinbase)
  with firstTree tree
```

```
...  | fstTreec nxTree₁ fstBlockc = addBlock
     (block→blockchain (blockc nxTree₁ fstBlockc (fstTree→coinbase fstBlock₁)))
     (blockc nxTree fstBlock₁ sndBlockCoinbase)
```

In this function, if the first transaction tree of the block is a genesis tree, it will return a blockchain of just one block. If it is a regular tree, it tries to find the first transaction tree of this block. Using a recursive definition of block to blockchain, it is possible to generate all the rest of this blockchain from this block.

It is not always possible to generate a block from the transaction tree. It is because the last transaction of a transaction tree must be a coinbase transaction. Here, the function that returns a decidable if it is possible to generate a block from the transaction tree.

```
txTree→Block : ∀
  {block time outputs totalFees qtTransactions}
  (tree : TXTree time block outputs totalFees qtTransactions)
  → Dec (RawBlock tree)
txTree→Block genesisTree =
  no λ{(rawBlockc (blockc _ _ sndBlockCoinbase)) → sndBlockCoinbase}
txTree→Block (txtree tree tx proofLessQtTX)
  with isCoinbaseTree (txtree tree tx proofLessQtTX)
...  | no ¬isCoinbase =
        no λ{ (rawBlockc (blockc _ _ coinbaseTree)) → ¬isCoinbase coinbaseTree}
...  | yes isCoinbase = let fTree = firstTree (txtree tree tx proofLessQtTX)
                            nxTree = fstTree.nxTree fTree
                            fBlock = fstTree.fstBlockc fTree
                        in yes (rawBlockc (blockc nxTree fBlock isCoinbase))
```

The definition of the raw block gets just the coinbase transaction tree as an explicit type. The other transaction tree can be founded opening the record.

```
record RawBlock
  {block : Nat}
  {time₂ : Time}
  {outputs₂ : List TXFieldWithId}
  {totalFees₂ : Amount}
  {qtTransactions₂ : tQtTxs}
  (tree₂ : TXTree time₂ block outputs₂ totalFees₂ qtTransactions₂)
  : Set where
  constructor rawBlockc
```

```
field
  {time}       : Time
  {outputs}    : List TXFieldWithId
  {totalFees}  : Amount
  {qtTransactions} : tQtTxs
  {tree}       : TXTree time block outputs totalFees qtTransactions
  rawBlock     : Block tree tree₂
```

The code of the definition of what is a coinbase tree:

```
coinbaseTree : ∀
  {block time outputs totalFees qtTransactions}
  (tree : TXTree time block outputs totalFees qtTransactions)
  → Set
coinbaseTree genesisTree = ⊥
coinbaseTree (txtree _ (normalTX _ _ _ _) _) = ⊥
coinbaseTree (txtree _ (coinbase _ _ _) _) = ⊤
```

The definition of a coinbase tree is the one that the last transaction is a coinbase.

The code verifies if the last transaction tree is a coinbase tree:

```
isCoinbaseTree : ∀
  {block time outputs totalFees qtTransactions}
  (tree : TXTree time block outputs totalFees qtTransactions)
  → Dec (coinbaseTree tree)
isCoinbaseTree genesisTree = no λ x → x
isCoinbaseTree (txtree _ (normalTX _ _ _ _) _) = no λ x → x
isCoinbaseTree (txtree _ (coinbase _ _ _) _) = yes tt
```

If it is, it returns that it is possible to create a block from that with the block definition. If it is not, it returns that it is impossible to create a block from this transaction tree.

But to create a block from this coinbase transaction tree, it is necessary to find the first tree of the block.

```
record fstTree
  {block : Nat}
  {time₂ : Time}
  {outputs₂ : List TXFieldWithId}
  {totalFees₂ : Amount}
```

```
{qtTransactions₂ : tQtTxs}
(txTree₂ : TXTree time₂ block outputs₂ totalFees₂ qtTransactions₂)
: Set where
constructor fstTreec
field
  {time}            : Time
  {outputs}         : List TXFieldWithId
  {totalFees}       : Amount
  {qtTransactions} : tQtTxs
  {tree}            : TXTree time block outputs totalFees qtTransactions
  nxTree            : nextTXTree tree txTree₂
  fstBlockc         : firstTreesInBlock tree
```

The definition of *fstTree* is that it has a tree that is before this tree in the type. And this tree before is the first in the block.

```
firstTreesInBlock : ∀
  {block time outputs totalFees qtTransactions}
  (tree : TXTree time block outputs totalFees qtTransactions)
  → Set
firstTreesInBlock genesisTree = ⊤
firstTreesInBlock (txtree genesisTree _ _) = ⊥
firstTreesInBlock (txtree (txtree _ (normalTX _ _ _ _) _) _ _) = ⊥
firstTreesInBlock (txtree (txtree _ (coinbase _ _ _) _) _ _) = ⊤
```

The decidable version of this *Set*:

```
isFirstTreeInBlock : ∀
  {block time outputs totalFees qtTransactions}
  (tree : TXTree time block outputs totalFees qtTransactions)
  → Dec (firstTreesInBlock tree)
isFirstTreeInBlock genesisTree = yes tt
isFirstTreeInBlock (txtree genesisTree (normalTX _ _ _ _) _) = no λ x → x
isFirstTreeInBlock (txtree genesisTree (coinbase _ _ _) _) = no λ x → x
isFirstTreeInBlock (txtree (txtree _ (normalTX _ _ _ _) _) _ _) = no λ x → x
isFirstTreeInBlock (txtree (txtree _ (coinbase _ _ _) _) _ _) = yes tt
```

In this case, it pattern match trees that are genesis tree or if the last transaction was a coinbase transaction.

```
firstTree : ∀
  {block time outputs totalFees qtTransactions}
```

```
      (tree : TXTree time block outputs totalFees qtTransactions)
      → fstTree tree
firstTree genesisTree = fstTreec (firstTX genesisTree) unit
firstTree {block₂} (txtree {block₁} tree tx proofLessQtTX)
  with isFirstTreeInBlock (txtree tree tx proofLessQtTX)
... | yes isFirst = fstTreec (firstTX (txtree tree tx proofLessQtTX)) isFirst
... | no ¬first with let fstTree = firstTree tree in block₂ == block₁
... | yes eq = let ftree = firstTree tree
                   nxTree = fstTree.nxTree ftree
                   fstBlock = fstTree.fstBlockc ftree
                   chgType = changeTXType nxTree fstBlock tx proofLessQtTX eq
               in TXChange.fTree chgType
... | no ¬eq = ⊥-elim impossible
      where postulate impossible : ⊥
```

To find the first tree in the block, there are two cases. The first case is that if the tree is a genesis tree, so the result is itself. The second case is if it a regular tree, so it still has to divide it in many cases. If this tree is already the first tree in the block, it will return itself. If this tree is not, it has to verify if the block number of the tree is the same as this tree. If the block number is equal, it can recursively find the first tree. If it is not, it has to provide proof that this tree must be the first and the blocks numbers are different.

To define what it means of one tree is next to another:

```
data nextTXTree :
  {block₁ : Nat}
  {time₁ : Time}
  {outputs₁ : List TXFieldWithId}
  {totalFees₁ : Amount}
  {qtTransactions₁ : tQtTxs}
  (txTree₁ : TXTree time₁ block₁ outputs₁ totalFees₁ qtTransactions₁)

  {block₂ : Nat}
  {time₂ : Time}
  {outputs₂ : List TXFieldWithId}
  {totalFees₂ : Amount}
  {qtTransactions₂ : tQtTxs}
  (txTree₂ : TXTree time₂ block₂ outputs₂ totalFees₂ qtTransactions₂)
  → Set where
```

firstTX : ∀ {*block time outputs totalFees qtTransactions*}
  (*txTree* : TXTree *time block outputs totalFees qtTransactions*)
  → nextTXTree *txTree txTree*
nextTX : ∀ {*block$_1$ time$_1$ outputs$_1$ totalFees$_1$ qtTransactions$_1$*}
  {*txTree$_1$* : TXTree *time$_1$ block$_1$ outputs$_1$ totalFees$_1$ qtTransactions$_1$*}

  {*block$_2$ time$_2$ outputs$_2$ totalFees$_2$ qtTransactions$_2$*}
  {*txTree$_2$* : TXTree *time$_2$ block$_2$ outputs$_2$ totalFees$_2$ qtTransactions$_2$*}

  (*nxTree* : nextTXTree *txTree$_1$ txTree$_2$*)

  {*outSize amount*}
  {*outputTX* : VectorOutput *time$_2$ outSize amount*}
  (*tx* : TX *txTree$_2$ outputTX*)
  (*proofLessQtTX* :
    Either
      (IsTrue (lessNat (finToNat *qtTransactions$_2$*) totalQtSub1))
      (isCoinbase *tx*))
  → nextTXTree *txTree$_1$* (txtree *txTree$_2$ tx proofLessQtTX*)

Both trees (*txTree1* and *txTree2*) can be equal or different. If both trees are the same, they are next to each other. If there is a proof that both trees are next to each other and if there is one tree that was generated from the last one, so the first tree is next to the last one.

4   Conclusion

Formal methods in cryptocurrency space are growing significantly. Companies like Input Output HK (IOHK), creator of Cardano, and Tezos are investing a lot in it. This work contributes to the formal specification and definition of Bitcoin.

A good way of defining Bitcoin is by creating a model of it in a language with dependent types. Agda looks like a good language for it, but other languages like CoQ, Lean can do this work too.

In this work, we define a lot of functionalities about Bitcoin. There were definitions of transactions, transactions tree, block, and blockchain. They were all defined in data constructors and records. Most of the model definition was in the transaction tree because of the state of Bitcoin changes after every transaction. There are other ways of doing the same thing, but I thought that this way is easier to define. Another way could be defining more characteristics of the block and blockchain in their types instead of doing all of it in the transaction tree (what was done in this work).

Some part of this code is not just for modeling the Bitcoin, but to validates inputs that can be wrong. They were functions that transform terms with simpler types (without dependent types) to terms with more complex types (with them). For example, transforming raw transactions into possible valid transactions.

4.1   Future work

In this work, there was a code that transforms a raw transaction into a possible valid transaction. It is not a decidable function, because there is no definition of what it is an invalid transaction. From future work, it should have a definition of what is an invalid raw transaction. So it will avoid that valid transaction will be discarded.

There is no definition of crypto functions like SHA-256 and elliptic curves in this work. One thing that can be done is importing these functions from some Agda or Haskell packages (*cryptohash* in *Hackage*).

In this cryptocurrency, there is no nonce and mining either.

This work does not have any IO operation. So it is not possible to add transactions in the blockchain from the command line or the network.

The cryptocurrency of this work does not have any smart contract. It would be good to define some of them in it.

References

Back, A., et al. (2002). Hashcash-a denial of service counter-measure.

Bastiaan, M. (2015). Preventing the 51%-attack: a stochastic analysis of two phase proof of work in bitcoin. In *Availab le at http://referaat. cs. utwente. nl/conference/22/paper/7473/preventingthe-51-attack-a-stochasticanalysis-oftwo-phase-proof-of-work-in-bitcoin. pdf.*

Beukema, W. (2014). Formalising the bitcoin protocol. In *21th twente student conference on it.*

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., ... others (2016). Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 acm workshop on programming languages and analysis for security* (pp. 91–96).

Chaudhary, K., Fehnker, A., Van De Pol, J., & Stoelinga, M. (2015). Modeling and verification of the bitcoin protocol. *arXiv preprint arXiv:1511.04173*.

Chaum, D. (1983). Blind signatures for untraceable payments. In *Advances in cryptology* (pp. 199–203).

Dai, W. (1998). B-money. *Consulted*, *1*, 2012.

Hopwood, D., Bowe, S., Hornby, T., & Wilcox, N. (2016). Zcash protocol specification. *Tech. rep. 2016–1.10. Zerocoin Electric Coin Company, Tech. Rep.*.

Kosba, A., Miller, A., Shi, E., Wen, Z., & Papamanthou, C. (2016). Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 ieee symposium on security and privacy (sp)* (pp. 839–858).

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 acm sigsac conference on computer and communications security* (pp. 254–269).

Melkonian, O. (2019). *Formalizing extended utxo and bitml calculus in agda* (Unpublished master's thesis).

Nakamoto, S., et al. (2008). Bitcoin: A peer-to-peer electronic cash system.

Noether, S. (2015). Ring signature confidential transactions for monero. *IACR Cryptology ePrint Archive*, *2015*, 1098.

Panurach, P. (1996). Money in electronic commerce: Digital cash, electronic fund transfer, and ecash. *Communications of the ACM*, *39*(6), 45–51.

Setzer, A. (2018). Modelling bitcoin in agda. *arXiv preprint arXiv:1804.06398*.

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, *58*(345-363), 5.

Wallace, B. (2011). The rise and fall of bitcoin. *Wired*, *19*(12).

Wood, G., et al. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, *151*, 1–32.