Fundação Getúlio Vargas

Modelagem Matemática

Programming a cryptocurrency in Agda

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

Contents

1 Introduction

1.1 Cryptocurrencies

In 1983, David Chaum created ecash (Panurach, 1996) , an anonymous cryptographic eletronic money. This cryptocurrency use RSA blind signatures (Chaum, 1983) to spend transactions. Later, in 1989, David Chaum found an eletronic 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 numerically value smaller than a defined target. So, to create a valide email, the sender (miner) has to spend a considerable CPU resource on it. Because, hash functions produces pratically random values, so the miner has to guess a lot of nonce values before find some nonce that make the hash of the email less than the target value. 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 propose 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 hashcash for its algorithm of proof of work.

In 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 open-source code. The identity of Satoshi Nakamoto is still unknown. Since that time, the total market of Bitcoin came to 330 billions dollars in 17 of December of 2018 when its value reached the historic peak of 20 thousands dollars.

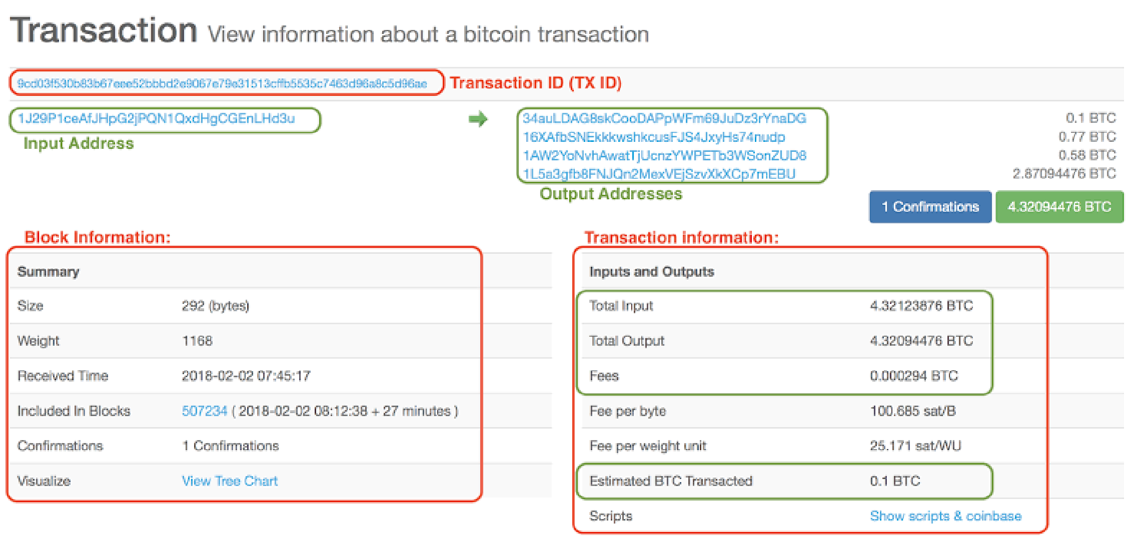
Other cryptocurrencies like Ethereum (Wood et al., 2014), Monero (Noether, 2015) and ZCash (Hopwood, Bowe, Hornby, & Wilcox, 2016) were created after Bitcoin, but Bitcoin is still the cryptocurrency with the biggest market value.

Ethereum is a cryptocurrency that uses account model instead of UTXO used in bitcoin for its transaction data struct. 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 is now transitioning from proof of work (used in Bitcoin) to proof of stake which will be the default proof mechanism of Ethereum 2.0 which is due to be released in 3 of January of 2020.

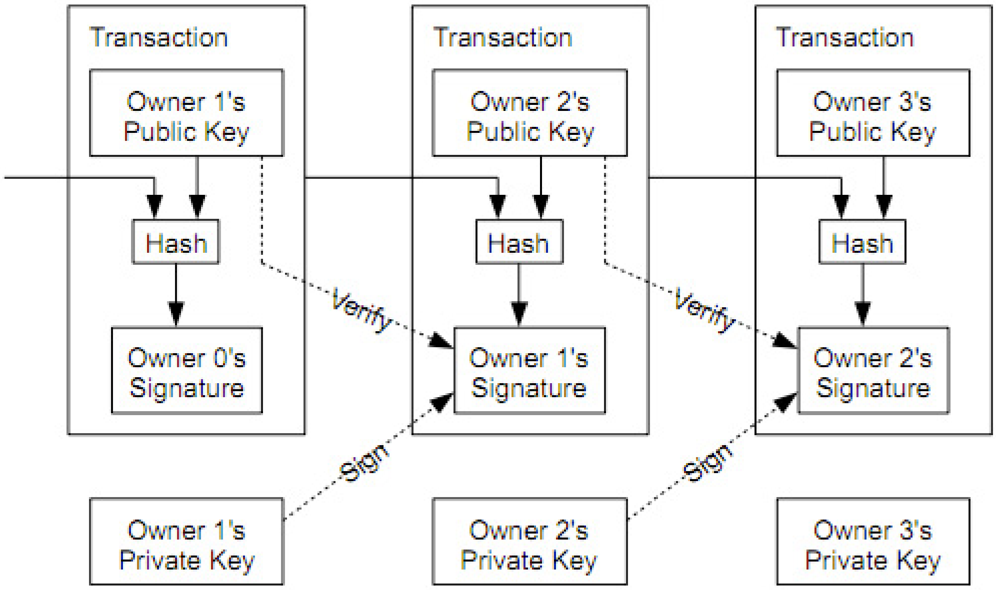
Monero and ZCash are both cryptocurrencies that focus on fungibility, privacy and descentralization. Monero uses an obfuscated public ledger, so anyone can send transactions, but nobody can tell the source, amount or destination. Zcash uses the concept of zero-knowledge proof called zk-SNARKs, which garantee privacy for its users.

1.2 Bitcoin

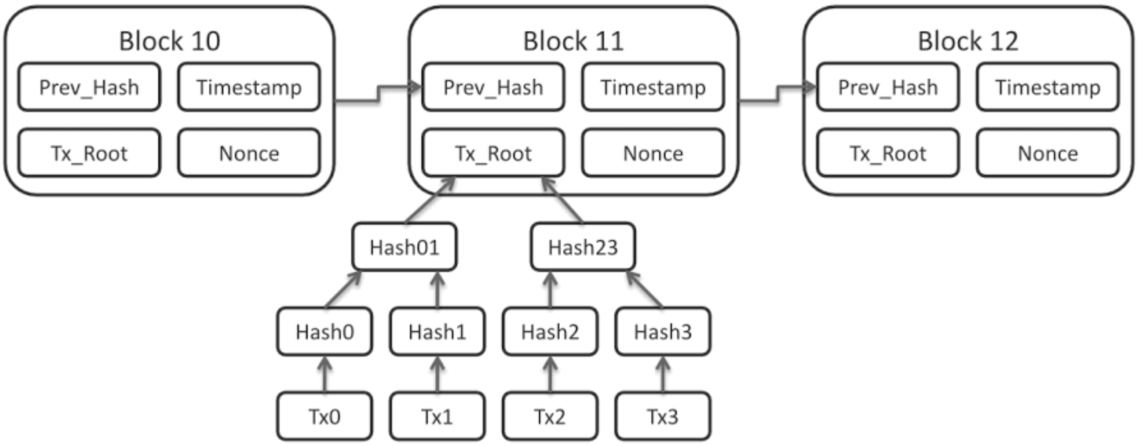
The bitcoin was made to be a peer to peer eletronic cash. It was made in one way that users can save and verify transactions without the need of a trusted party. Because of that no authority or goverment can block the bitcoin.



Transactions in bitcoins are an array of input of previous transactions and an array of outputs. The mining transaction does require an input. For each input of the transaction, it is necessary a signature signed with a private key to prove the ownership of the bitcoins.



Transactions are grouped in a block. 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 (an chain of blocks). Each block should be created in an average of 10 minutes. This time was choosen because 10 minutes is enough to propagate the block throughout the world. To make the blockchain tamper-proof, there is a concept called proof of work in Bitcoin. So the miner has choose a random value as nonce that makes the hash of the block less than a certain value. This value is choosen in a way that each block should be generated in 10 minutes in average. If the value is too low, miners will take more time to find a nonce that make the hash block less than the it. If it is too high, it will be easier to find a nonce and they will find it faster.

When two blocks are mined in nearly the same time, there are two valid blockchains. It is because the last block in the both blockchains are valid but different. Because of this problem, in 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.

In Bitcoin, there is a possibility of 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 become the valid chain, because it is bigger. So all transactions from previous blockchain from 46 to 50 blocks become invalid. Because of that, when someone make 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.

1.3 Ethereum

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

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

In Bitcoin, the creator of the contract as to pay the amount proportional to its size. In Ethereum, it is different, there is a concept of gas. Each smart contract in Ethereum is made by a serie of instructions. Each instruction consume different computional effort. Because of that, in Ethereum, there is a concept of gas, that measure how much computional 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 computional effort 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 payed is not high enough, the contract will not be executed, because there are some other contracts that pay more that will be executed instead of this one.

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

1.4 Introduction

Before this work, there were some research in this field. Antom Setzel (Setzer, 2018) already code the definitions of transactions and transactions tree of bitcoin. Orestis Melkonian start to formalize Bitcoin Script.

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

1.5 Agda Introduction

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

1.6 Syntax

In Agda, Set is equal to type. In dependent type languages, it is possible to create a function that return a type.

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

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

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

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

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

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

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
```

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

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

Implicits arguments are inside . In case of the function id, the type of the input can be deduced by the compiler. For example, the only type that zero can be is Natural.

```
zeroℕ : ℕ
zeroℕ = id zero
```

1.7 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 conversions α-conversion and β-reduction. In α-conversion, $(\lambda x.M[x]) \to (\lambda y.M[y])$. So in every free variable in M will be renamed from x 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 bind outside. For example, $((\lambda x.x)x)$. The blue x is binded for the green x, but the red x is not binded 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 : Set} → A → A
id = λ x → x
```

The simplest function is the identity function made in Agda

```
id' : {A : Set} → A → A
id' x = x
```

This is another way of writing the same function.

```
true : {A : Set} → A → A → A
true x y = x

false : {A : Set} → A → A → A
false x y = y
```

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

```
zero : {A : Set} → (A → A) → A → A
zero suc z = z

one : {A : Set} → (A → A) → A → A
one suc z = suc z

two : {A : Set} → (A → A) → A → A
two suc z = suc (suc z)
```

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

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

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

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

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

$$\begin{aligned}
&\mathsf{plus} : \{A : \mathsf{Set}\} \to ((A \to A) \to A \to A)\\
&\quad \to ((A \to A) \to A \to A)\\
&\quad \to ((A \to A) \to A \to A)\\
&\mathsf{plus}\ n\ m = \lambda\ \mathit{suc}\ z \to n\ \mathit{suc}\ (m\ \mathit{suc}\ z)
\end{aligned}$$

$$\begin{aligned}
&_+_ : \{A : \mathsf{Set}\} \to ((A \to A) \to A \to A)\\
&\quad \to ((A \to A) \to A \to A)\\
&\quad \to ((A \to A) \to A \to A)\\
&_+_\ n\ m\ \mathit{suc}\ z = n\ \mathit{suc}\ (m\ \mathit{suc}\ z)
\end{aligned}$$

Plus is defined this way using lambda calculus.

$$\mathsf{one{+}one} : \{A : \mathsf{Set}\} \to \mathsf{Result}\ (\_+\_\ \{A\}\ \mathsf{one}\ \mathsf{one})$$
$$\mathsf{one{+}one} = \mathsf{res}\ (\lambda\ \mathit{suc}\ z \to \mathit{suc}\ (\mathit{suc}\ z))$$

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

$$\mathsf{emptyList} : \{A\ \mathit{List} : \mathsf{Set}\} \to (A \to \mathit{List} \to \mathit{List}) \to \mathit{List} \to \mathit{List}$$
$$\mathsf{emptyList}\ \_{::}\_\ \mathit{nil} = \mathit{nil}$$

$$\mathsf{natList} : \{A\ \mathit{List} : \mathsf{Set}\} \to (((A \to A) \to A \to A) \to \mathit{List} \to \mathit{List}) \to \mathit{List} \to \mathit{List}$$
$$\mathsf{natList}\ \_{::}\_\ \mathit{nil} = \mathsf{one}\ {::}\ (\mathsf{two}\ {::}\ \mathit{nil})$$

This is how lists are defined in Lambda Calculus

$$\mathsf{sumList} : \{A\ \mathit{List} : \mathsf{Set}\} \to \mathsf{Result}\ (\mathsf{natList}\ \{A\}\ \{(A \to A) \to A \to A\}\ \_+\_\ \mathsf{zero})$$
$$\mathsf{sumList} = \mathsf{res}\ (\lambda\ \mathit{suc}\ z \to \mathit{suc}\ (\mathit{suc}\ (\mathit{suc}\ z)))$$

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

$\mathsf{left} : \{A\ B\ C : \mathsf{Set}\} \rightarrow A \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$
$\mathsf{left}\ x\ f\ g = f\ x$

$\mathsf{right} : \{A\ B\ C : \mathsf{Set}\} \rightarrow B \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$
$\mathsf{right}\ x\ f\ g = g\ x$

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.

$\mathsf{zero\text{-}left} : \{A\ B\ C : \mathsf{Set}\} \rightarrow (((A \rightarrow A) \rightarrow A \rightarrow A) \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$
$\mathsf{zero\text{-}left} = \mathsf{left\ zero}$

$\mathsf{one\text{-}left} : \{A\ B\ C : \mathsf{Set}\} \rightarrow (((A \rightarrow A) \rightarrow A \rightarrow A) \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$
$\mathsf{one\text{-}left} = \mathsf{left\ one}$

$\mathsf{false\text{-}right} : \{A\ B\ C : \mathsf{Set}\} \rightarrow (A \rightarrow C) \rightarrow ((B \rightarrow B \rightarrow B) \rightarrow C) \rightarrow C$
$\mathsf{false\text{-}right} = \mathsf{right\ false}$

$\mathsf{true\text{-}right} : \{A\ B\ C : \mathsf{Set}\} \rightarrow (A \rightarrow C) \rightarrow ((B \rightarrow B \rightarrow B) \rightarrow C) \rightarrow C$
$\mathsf{true\text{-}right} = \mathsf{right\ true}$

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

$\mathsf{zero\text{-}isZero} : \{A : \mathsf{Set}\} \rightarrow \mathsf{Result}\ (\mathsf{zero\text{-}left}\ \{A\}\ \mathsf{isZero\ id})$
$\mathsf{zero\text{-}isZero} = \mathsf{res}\ (\lambda\ \mathit{true}\ \mathit{false} \rightarrow \mathit{true})$

$\mathsf{one\text{-}isZero} : \{A : \mathsf{Set}\} \rightarrow \mathsf{Result}\ (\mathsf{one\text{-}left}\ \{A\}\ \mathsf{isZero\ id})$
$\mathsf{one\text{-}isZero} = \mathsf{res}\ (\lambda\ \mathit{true}\ \mathit{false} \rightarrow \mathit{false})$

$\mathsf{false\text{-}id} : \{A : \mathsf{Set}\} \rightarrow \mathsf{Result}\ (\mathsf{false\text{-}right}\ \{(A \rightarrow A) \rightarrow A \rightarrow A\}\ \mathsf{isZero\ id})$
$\mathsf{false\text{-}id} = \mathsf{res}\ (\lambda\ \mathit{true}\ \mathit{false} \rightarrow \mathit{false})$

$\mathsf{true\text{-}id} : \{A : \mathsf{Set}\} \rightarrow \mathsf{Result}\ (\mathsf{false\text{-}right}\ \{(A \rightarrow A) \rightarrow A \rightarrow A\}\ \mathsf{isZero\ id})$
$\mathsf{true\text{-}id} = \mathsf{res}\ (\lambda\ \mathit{true}\ \mathit{false} \rightarrow \mathit{false})$

Either is usefull when defining one function that works for left and another that works for the right. The function choosen for left was if a natural number is zero and the function choosen for right was if the identity function.

$\mathsf{tuple} : \{A\ B\ C : \mathsf{Set}\} \rightarrow A \rightarrow B \rightarrow (A \rightarrow B \rightarrow C) \rightarrow C$
$\mathsf{tuple}\ x\ y\ f = f\ x\ y$

This way is how tuple is defined in Lambda Calculus.

$\text{zero-false} : \{A\ B\ C : \text{Set}\} \to (((A \to A) \to A \to A) \to (B \to B \to B) \to C) \to C$
$\text{zero-false} = \text{tuple zero false}$

$\text{one-true} : \{A\ B\ C : \text{Set}\} \to (((A \to A) \to A \to A) \to (B \to B \to B) \to C) \to C$
$\text{one-true} = \text{tuple one true}$

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

$\text{add-true} : \{A : \text{Set}\} \to ((A \to A) \to A \to A) \to (A \to A \to A) \to ((A \to A) \to A \to A)$
$\text{add-true}\ n\ b\ suc\ z = b\ (suc\ (n\ suc\ z))\ (n\ suc\ z)$

$\text{add-zero-false} : \{A : \text{Set}\} \to \text{Result}\ (\text{zero-false}\ \{(A \to A) \to A \to A\}\ \text{add-true})$
$\text{add-zero-false} = \text{res}\ (\lambda\ suc\ z \to z)$

$\text{add-one-true} : \{A : \text{Set}\} \to \text{Result}\ (\text{one-true}\ \{(A \to A) \to A \to A\}\ \text{add-true})$
$\text{add-one-true} = \text{res}\ (\lambda\ suc\ z \to suc\ (suc\ z))$

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

1.8 Martin-Löf type theory

Agda also provides a proof assistance based on intentional Martin-Löf type theory.

1.9 Types

In Martin-Löf type theory, there are 3 finite types and 5 constructors types. The 0 type contain 0 terms, it is called empty type and it is written bot.

The 1 type is the type with just 1 canonical term and it represents existence. It is called unit type and it is written top .

The 2 type contains 2 canonical terms. It represents a choice between two values.

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

If statement is defined using booleans

1.10 Types Constructors

The sum-types contain an ordered pair. The second type can depend on the first type. It has the same meaning of exist.

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

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

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

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 structs like linked list of natural numbers, trees, graphs are 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.

It looks like CoQ, but does not have tatics. Agda is a total language, so it is garanteed that the code always terminal and coverage all inputs. Agda needs it to be a consistent language.

Agda has inductive data types that are similar to algebric data types in non-depently typed programming language. The definition of Peano numbers in Agda:

```
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 some restrictions in types that is not possible in other language. For example, get the first element of a vector. For it, it is necessary to specify in the type that the vector should have at size greater or equal tha than one.

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

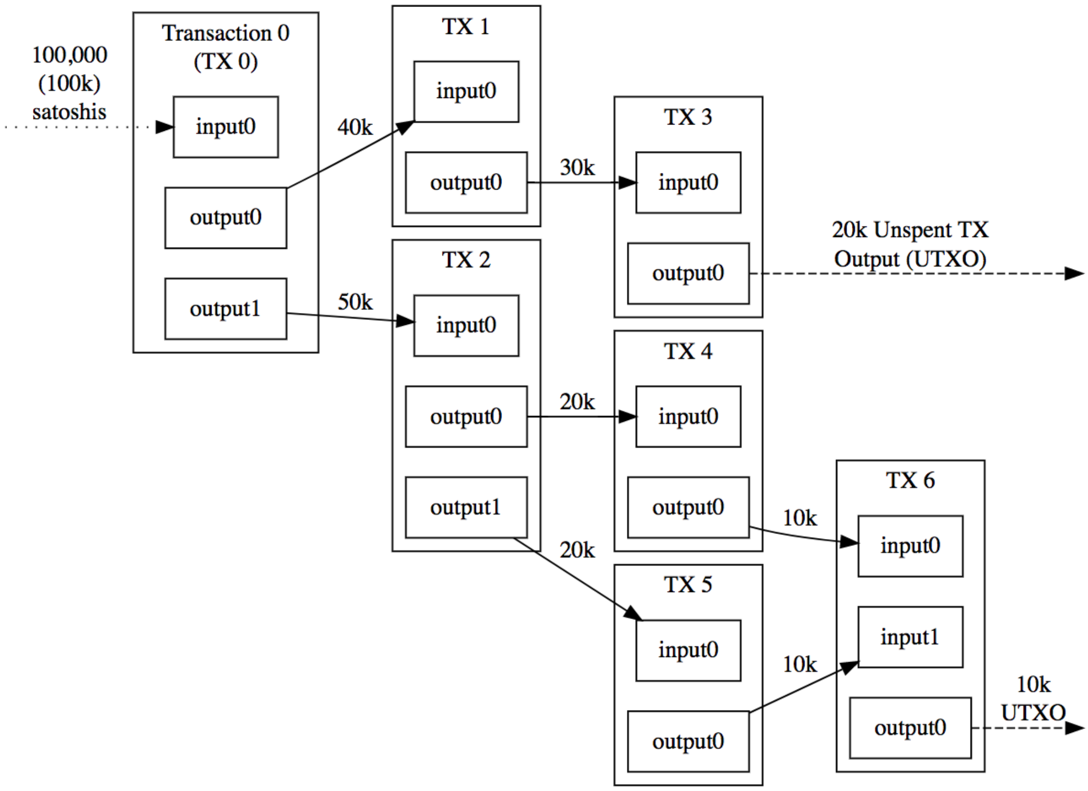
Another good example is that in sum of two matrices, they should have the same dimentions.

```
_+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)
```

1.11 UTXO Bitcoin

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

In account model, it is saved the address and the balance of each address. For example, the data struct 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 of balance each address has, but it is not possible to know how they got in this state.



Triple-Entry Bookkeeping (Transaction-To-Transaction Payments) As Used By Bitcoin

In UTXO model, each transaction is saved in the transaction tree. Every transaction is composed of multiples inputs and multiples outputs. But all inputs have to never been spent before.

Because of that, in UTXO model, it is easy to make a new transaction from previous one, but it is harder to know how much each one has. The wallet that calculate how much balance each address has.

In account model, there could be one kind of vulnerability that is less probabable to happen in UTXO model. Because there is an undesirable intermediary state that there is some address without balance while another has not already received his money.

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

In account model, it is straight foward to know how much balance each address has.

In UTXO model, this calculation is made offchain. It can be a good thing, because each user has more privacy.

1.12 TXTree in Agda

2 Methods

2.1 Cripto Functions

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

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

Public key is generated from private key. But getting private key from public key is impossible. 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 collision is very low). The function is used from a big domain to a small domain. For example, a hash of big file (some GBs) is an integer of just some bytes. It is very usefull to prove for example that 2 files are equal. If the hash of two files are equal, so the files are equal. 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 theses. 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
```

2.2 Transactions

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

```
data VectorOutput : (time : Time) (size : Nat) (amount : Amount) → Set where
  el : ∀ {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. 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 spend in this transaction,

elStart is a proof that the position of TXFieldWithId is the last one. It is used after to specify wich 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 a proof that the total amount of money in all inputs are bigger than the total amount of outputs. The remainder will be used by the miner.

2.3 Transaction Tree

Transaction tree is one of most important data structs in bitcoin. In the transaction tree, there are all unspent transaction outputs (UTXO). In every new transaction, the UTXOs used as input is removed from 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 the transactions in TXTree. Block is related in which block the transaction tree is. After every new coinbase transaction (the miner transaction), the block size increment in one quantity. Total fees are how much the miner will have in fee of transactions if he makes a block with these transactions. Quantity of transactions 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 has a number maximum 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 crypto currency 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 cripto currency. There are two kinds of transaction. Coinbase transaction is the transaction done by the miner. In coinbase, they have just outputs and do not have any input. pAmountFee is a proof that the output of 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 receive 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 a proof that the output amount is greather 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 return 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 {block} (normalTX _ _ _ _) = block
nextBlock {block} (coinbase _ _ _) = suc block
```

If it is a normal transaction, the block continue the same. If it is a coinbase transaction, the next transaction 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 quantity of transaction in the block. It has to receive a proof that the quantity of transaction that was before this new transaction was less than then maximum quantity of transaction allowed. So it is guaranteed that the quantity of transactions will never be greater than the maximum allowed. If it is a coinbase transaction, it will be a new block. So the quantity of transactions start 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
```

IncFee is a function that increment 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 trasaction 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
```

outFee+RewardBlock is a proof that the amount of output transactions is equal to total fees of others transactions plus the block reward.

3 Conclusion

References

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

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.*.

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*.

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.