Uma versão simplificada do Bitcoin, implementada em Agda

Guilherme Horta Alvares da Silva
Orientador: Flávio Codeço Coelho

Fundação Getulio Vargas

2019

Uma versão simplificada do Bitcoin, implementada em Agda

- Objetivo
- Justificativa
- Introdução
  - Criptomoedas
  - Agda
  - Bugs em criptomoedas
- Trabalho executado
- Próximos passos
- Referências Bibliográficas

Objetivo

- Programar uma criptomoeda (similar ao Bitcoin) em Agda, que é uma linguagem com tipos dependentes.

Justificativa

- Programar um protocolo de criptomoedas livre de erros (bugs).
- Utilizar Agda permite, além da programação da criptomoeda, especificar como ela deve funcionar (Norell, 2008).

Linguagem Funcional

- Em linguagens funcionais, toda função retorna a mesma saída para a mesma entrada.
- Em Agda, toda função é total e ela sempre termina.

Efeitos Colaterais

- Para Agda ser utilizada como um programa, ela tem que lidar com o sistema operacional. Operação chamada de IO (Input Output).
- Várias pesquisas estão sendo feitas para minimizar os efeitos colaterais do IO.
- Lógica linear é usada para lidar com recursos, como abrir arquivos.
- Efeitos algébricos são utilizados para distinguir os diferentes tipos de efeitos relacionados ao sistema operacional.

Efeitos Colaterais

- Frontend e backend de serviços estão sendo programados na mesma linguagem funcional para minimizar os efeitos colaterais.
- Haxl, usado no Facebook, é uma ferramenta que faz o cache das operações IO para fácil debug, log e teste.
- Facebook também faz Hot Swapping de forma funcional. Ou seja, ele não reinicia o processo para iniciar um novo. Algo que já era comum em Lisp.
- Sistemas distribuídos já estão sendo completamente descritos por meio de linguagens funcionais. Whatsapp utiliza Elixir como linguagem.

Benefícios

- Por causa dos inúmeros benefícios das linguagens funcionais, elas estão sendo cada vez mais utilizadas na indústria.
- Garantem corretude de software. É de fácil reprodutibilidade, testabilidade e é mais fácil de debugar.
- Entretanto, elas são mais difíceis de serem aprendidas. Necessitam alto grau de abstração. Portanto, é mais difícil de encontrar mão de obra que saiba utilizá-la.

Empresas

Jane Street

Criptomoedas

Agda

- Linguagem funcional com sistema de tipos expressivos capazes de representar as especificações.
- Possibilita especificar e programar em um único lugar.
- O processo de verificação acontece no compilador.

Agda

- A linguagem não possui *Built-in* como em Python.
- Tipos como inteiros, ponto flutuantes, *strings* e vetores devem ser definidos pelo próprio usuário.
- Tipos em Agda são uma generalização de tipos de dados algébricos encontrados em Haskell e ML.

Identidade

- Agda utiliza alguns conceitos do Lambda Calculus:

$$\begin{aligned} &\mathsf{id} : \{A : \mathsf{Set}\} \to A \to A \\ &\mathsf{id} = \lambda\ x \to x \end{aligned}$$

$$\begin{aligned} &\mathsf{id'} : \{A : \mathsf{Set}\} \to A \to A \\ &\mathsf{id'}\ x = x \end{aligned}$$

Booleanos

- Definição de booleanos:

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

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

Números naturais

- Definição de números naturais:

$$\begin{aligned}&\mathsf{zero} : \{A : \mathsf{Set}\} \to (A \to A) \to A \to A\\&\mathsf{zero}\ \mathit{suc}\ z = z\end{aligned}$$

$$\begin{aligned}&\mathsf{one} : \{A : \mathsf{Set}\} \to (A \to A) \to A \to A\\&\mathsf{one}\ \mathit{suc}\ z = \mathit{suc}\ z\end{aligned}$$

$$\begin{aligned}&\mathsf{two} : \{A : \mathsf{Set}\} \to (A \to A) \to A \to A\\&\mathsf{two}\ \mathit{suc}\ z = \mathit{suc}\ (\mathit{suc}\ z)\end{aligned}$$

Zero

```
isZero : {A : Set} → ((A → A) → A → A) → (A → A → A)
isZero n true false = n (λ _ → false) true

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

isZero-two : {A : Set} → Result (isZero {A} two)
isZero-two = res (λ true false → false)
```

Soma

```
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)
```

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

Lista

```
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)

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

Soma

- Definição de tipos de soma:

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

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

Tipo soma

```
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)
```

Tuplas

- Definição de tupla:

$$\begin{aligned}&\text{tuple} : \{A\ B\ C : \text{Set}\} \to A \to B \to (A \to B \to C) \to C\\&\text{tuple}\ x\ y\ f = f\ x\ y\end{aligned}$$

$$\begin{aligned}&\text{zero-false} : \{A\ B\ C : \text{Set}\} \to (((A \to A) \to A \to A)\\&\quad \to (B \to B \to B) \to C) \to C\\&\text{zero-false} = \text{tuple zero false}\end{aligned}$$

$$\begin{aligned}&\text{one-true} : \{A\ B\ C : \text{Set}\} \to (((A \to A) \to A \to A)\\&\quad \to (B \to B \to B) \to C) \to C\\&\text{one-true} = \text{tuple one true}\end{aligned}$$

Adição de tuplas

```
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))
```

Agda

- Definição de números naturais em Agda:
  - data $\mathbb{N}$: Set where
    zero : $\mathbb{N}$
    suc : $\mathbb{N} \rightarrow \mathbb{N}$

Agda

- Eliminação dos números naturais em Agda:

  - $\mathbb{N}\text{-elim} : (target : \mathbb{N})\ (motive : (\mathbb{N} \to \text{Set}))$
    $(base : motive\ \text{zero})$
    $(step : (n : \mathbb{N}) \to motive\ n \to motive\ (\text{suc}\ n))$
    $\to motive\ target$
    $\mathbb{N}\text{-elim}\ \text{zero}\ motive\ base\ step = base$
    $\mathbb{N}\text{-elim}\ (\text{suc}\ target)\ motive\ base\ step =$
    $step\ target\ (\mathbb{N}\text{-elim}\ target\ motive\ base\ step)$

Agda

- Definição de isomorfismo:
  - record $\_\simeq\_$ $(A\ B : \mathsf{Set}) : \mathsf{Set}$ where
    field
      $\mathsf{to} : A \to B$
      $\mathsf{from} : B \to A$
      $\mathsf{from} \circ \mathsf{to} : \forall\ (x : A) \to \mathsf{from}\ (\mathsf{to}\ x) \equiv x$
      $\mathsf{to} \circ \mathsf{from} : \forall\ (y : B) \to \mathsf{to}\ (\mathsf{from}\ y) \equiv y$

Agda

- Definição de número binário:
  - data Bit : Set where
    0# : Bit
    1# : Bit

    Bin^+ : Set
    Bin^+ = List Bit

    data Bin : Set where
    0# : Bin
    1#_ : (bs : Bin^+) → Bin

    postulate Bin≃ℕ : Bin ≃ ℕ

Agda

- Adição usando a recursão:
  - $\_+'\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
    $\text{zero} +' m = m$
    $\text{suc}\ n +' m = \text{suc}\ (n + m)$

Agda

- Adição usando a eliminação dos naturais:
  - $\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
    $n + m = \mathbb{N}\text{-elim}\ n\ (\lambda\ \_ \to \mathbb{N})\ m\ \lambda\ \_\ v \to \text{suc}\ v$

Agda

- Eliminação do tipo vazio:
  - data $\bot$: Set where

    $\bot$-elim : $\{A : \text{Set}\}$ $(bot : \bot) \rightarrow A$
    $\bot$-elim ()

Agda

```
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
```

Agda

- Tipo booleano em Agda:
  - $\mathsf{Bool} : \mathsf{Set}$
    $\mathsf{Bool} = \mathsf{Either}\ \top\ \top$

Agda

- Usando tipos booleanos em casos:
  - if_then_else_ : $\{l : \text{Level}\}\ \{A : \text{Set}\ l\}$
    $(b : \text{Bool})\ (tRes\ fRes : A) \rightarrow A$
    $\text{if}\ b\ \text{then}\ tRes\ \text{else}\ fRes =$
    $\text{Either-elim}\ b\ (\lambda\ \_ \rightarrow tRes)\ \lambda\ \_ \rightarrow fRes$

Agda

- Eliminação do para todo:
  - $\forall$-elim : $\forall$ $\{A : \mathsf{Set}\}$ $\{B : A \to \mathsf{Set}\}$
    $(L : \forall\ (x :\ A) \to B\ x)$
    $(M : A)$

    $\to B\ M$
    $\forall$-elim $L\ M = L\ M$

Agda

- Tipo soma:
  - data $\sum$ $(A : \text{Set})$ $(B : A \to \text{Set}) : \text{Set}$ where
    $\langle \_ , \_ \rangle : (x : A) \to B\ x \to \sum\ A\ B$

    $\sum$-elim : $\forall\ \{A : \text{Set}\}\ \{B : A \to \text{Set}\}\ \{C : \text{Set}\}$
    $\to (\forall\ x \to B\ x \to C)$
    $\to \sum\ A\ B$

    $\to C$
    $\sum$-elim $f \langle\ x\ ,\ y\ \rangle = f\ x\ y$

Agda

- Tipo vetor:
  - data Vec : $\mathbb{N} \rightarrow$ Set where
    [] : Vec zero
    _::_ : $\{size : \mathbb{N}\} \rightarrow \mathbb{N} \rightarrow$ Vec *size* $\rightarrow$ Vec (suc *size*)

```
nil : Vec zero
nil = []

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

Agda

- Encontrando primeiro elemento do vetor:
  - $\mathsf{head} : \{A : \mathsf{Set}\}\ \{n : \mathbb{N}\}\ (\mathit{vec} : \mathsf{Vector}\ A\ (\mathsf{suc}\ n)) \to A$
    $\mathsf{head}\ (x :: \mathit{vec}) = x$

Agda

- Tipos booleanos:
  - data Boolean : Set where
    true : Boolean
    false : Boolean

Agda

- Funções que retornam tipo:
  - $\mathsf{boolean{\to}Set} : (b : \mathsf{Boolean}) \to \mathsf{Set}$
    $\mathsf{boolean{\to}Set\ true} = \mathbb{N}$
    $\mathsf{boolean{\to}Set\ false} = \mathsf{Bool}$

Agda

- Utilizando records:

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

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

Agda

- Demonstrando que elementos podem ser encontrados:

```
module WF {A : Set} (_<_ : Rel A) where
  data Acc (x : A) : Set where
    acc : (∀ y → y < x → Acc y) → Acc x

  Well-founded : Set
  Well-founded = ∀ x → Acc x
```

Agda

- Definindo o que é uma categoria:

```
record Category (C : Set → Set → Set) : Set₁ where
  constructor cat
  field
    idc : {a : Set} → C a a
    comp : {a b c : Set} → C a b → C b c → C a c

catFunc : Category λ x y → (x → y)
catFunc = cat id _∘_
```

Definição

- Uma criptomoeda é um meio de troca descentralizado que se utiliza da tecnologia de blockchain e da criptografia para assegurar a validade das transações e a criação de novas unidades da moeda.
- O Bitcoin é considerado a primeira moeda digital mundial descentralizada, constituindo um sistema econômico alternativo (*peer-to-peer electronic cash system*) e responsável pelo ressurgimento do sistema bancário livre. (Nakamoto et al., 2008)
- O Bitcoin permite transações financeiras sem intermediários, mas verificadas por todos usuários (nodos da rede). Essas transações são gravadas em um banco de dados distribuídos (uma rede descentralizada), chamado de *blockchain*.

Maleabilidade de transação

- Nesse tipo de bug, é possível alterar o hash da transação depois que ela foi enviada.
- Todos os dados para calcular o hash não eram previamente calculados. Assim, o minerador poderia alterar o hash da transação.
- O ataque consistiria de um usuário enviar uma transação e ela não ser confirmada pelo sistema. Logo em seguida, esse mesmo usuário enviaria uma outra transação. Dessa forma, ele faria duas transações com a mesma moeda.
- Esse tipo de BUG pode ser evitado usando tipos dependentes. Colocando como característica da transação, o fato de seu ID ser único.

DAO Bug

- *Bug* que aconteceu em um cripto-contrato da rede Ethereum com um prejuízo de mais do que 250 milhões de dólares. (Wood et al., 2014)
- No cripto-contrato, existia uma função recursiva que não terminava. Ou seja, o usuário enviava uma quantidade de ethereum, depois acontecia um *loop* infinito e só depois era feito a atualização do seu balanço.
- Em Agda, esse tipo de bug seria evitado, pois é necessário provar que a função termina. Logo, *loops* infinitos não são possíveis em Agda.

Função hash

- Uma função hash é uma função que serve para comprimir dados grandes, como um vídeo, em um número pequeno. De forma que dois arquivos diferentes sempre terão hashes diferentes (injetividade).
- Pelo princípio da casa dos pombos, isso é impossível. Porém, nunca foi encontrado dois arquivos que possuem o mesmo hash no SHA-256.
- O Google encontrou dois arquivos que possuíam o mesmo hash no SHA-1. Logo, em seguida, essa função hash parou de ser usada.

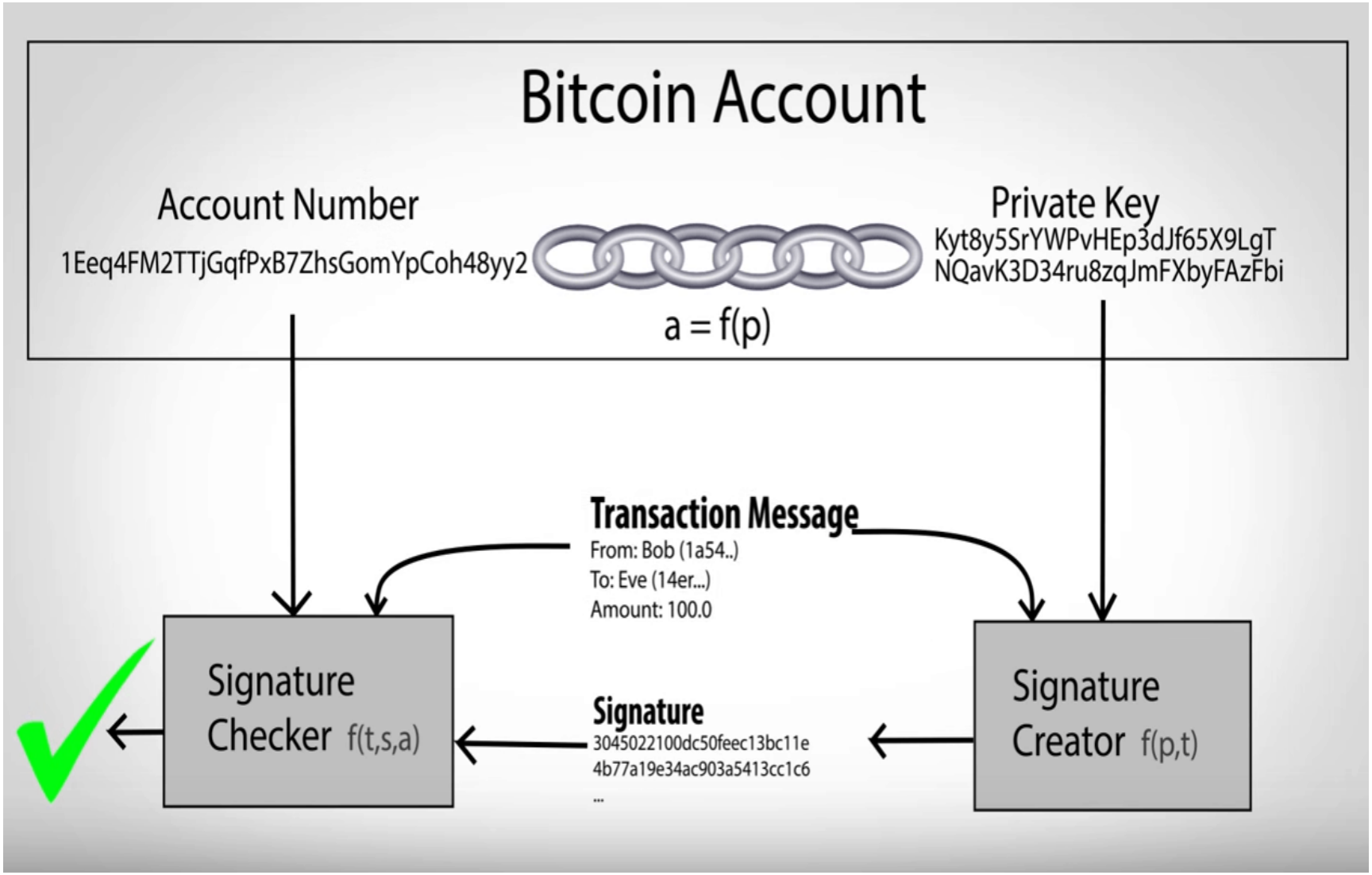
Funções criptográficas

- No Bitcoin, existe a chave privada e a pública.
- A chave privada serve para assinar a transação.
- A chave pública, derivada da chave privada, serve para demonstrar que você é o dono da chave privada.
- Essas funções utilizam como base a função hash SHA-256.

Funções criptográficas

```
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
```

Conta do Bitcoin



Transações

- A partir de transações, é possível enviar Bitcoins de uma conta para outra.
- Transações são como um cheque. O indivíduo especifica um valor e assina a transação.
- Na transação do Bitcoin, as transações anteriormente não gastas devem ser especificadas. No caso da transação do minerador, isso não deve ser especificado. Além do mais, deve ser especificado quem deve receber o dinheiro dessas transações e também a transação deve ter uma assinatura gerada a partir da chave privada comprovando que o usuário da chave pública aceitou fazer aquela transação.

Transações

```
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
```

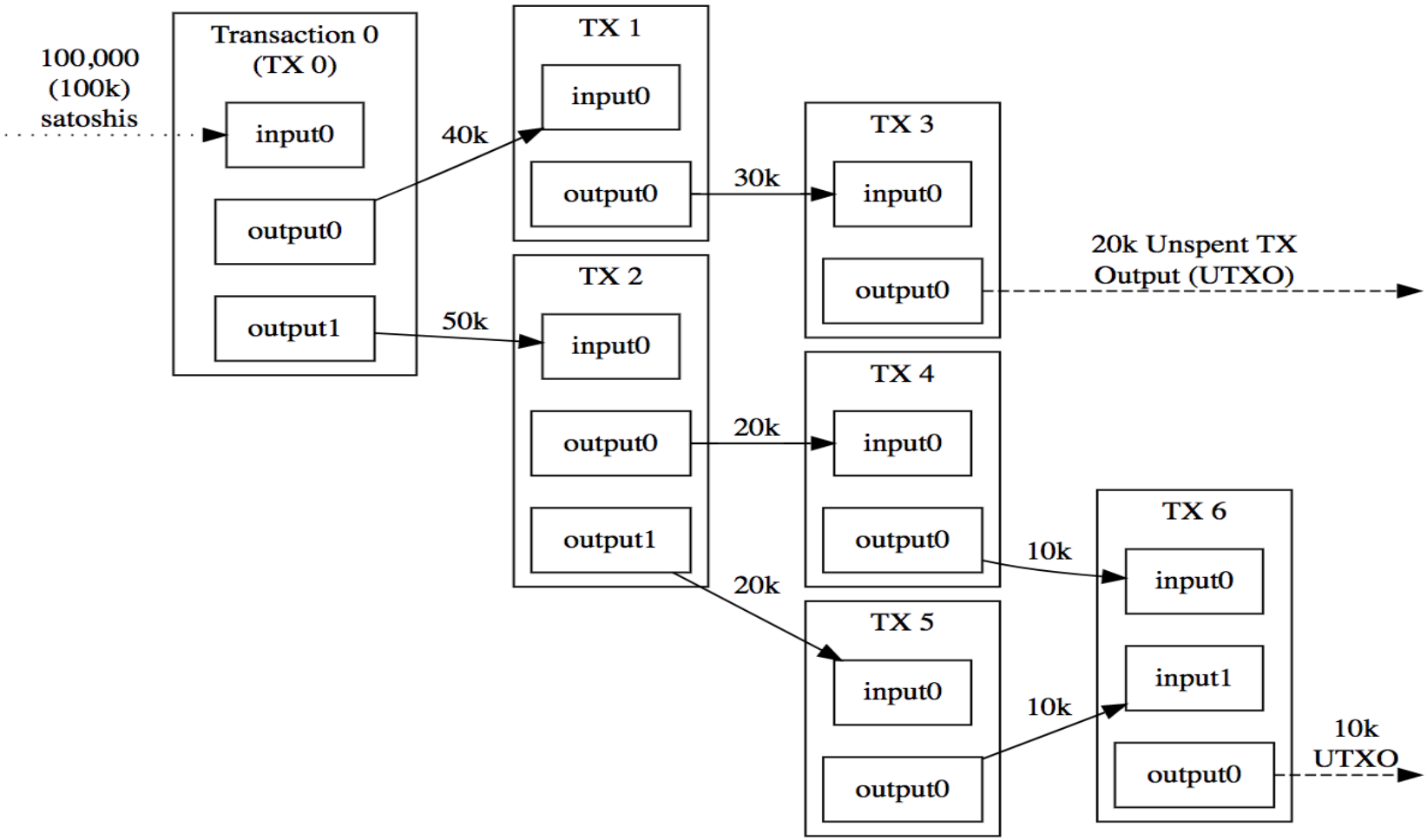
Modelo de transações

- O Bitcoin utiliza o modelo de transações não gastas enquanto o Ethereum utiliza o tradicional modelo bancário.
- No modelo bancario, cada conta possui um saldo. E em toda transação, o saldo da pessoa que enviou é subtraído e o saldo da pessoa que recebeu é incrementado.
- Nesse modelo tradicional, é fácil verificar o saldo de cada um. Porém é difícil saber como se chegou nesse estado final.

Modelo de transações não gastas

- No modelo de transações não gastas, toda transação é adicionada à árvore de transações.
- Para saber o saldo de uma conta é necessário olhar todas as transações não gastas enviadas para essa conta.
- Para transacionar a moeda é necessário utilizar como entrada as saídas das outras transações não gastas.

Bitcoin UTXO

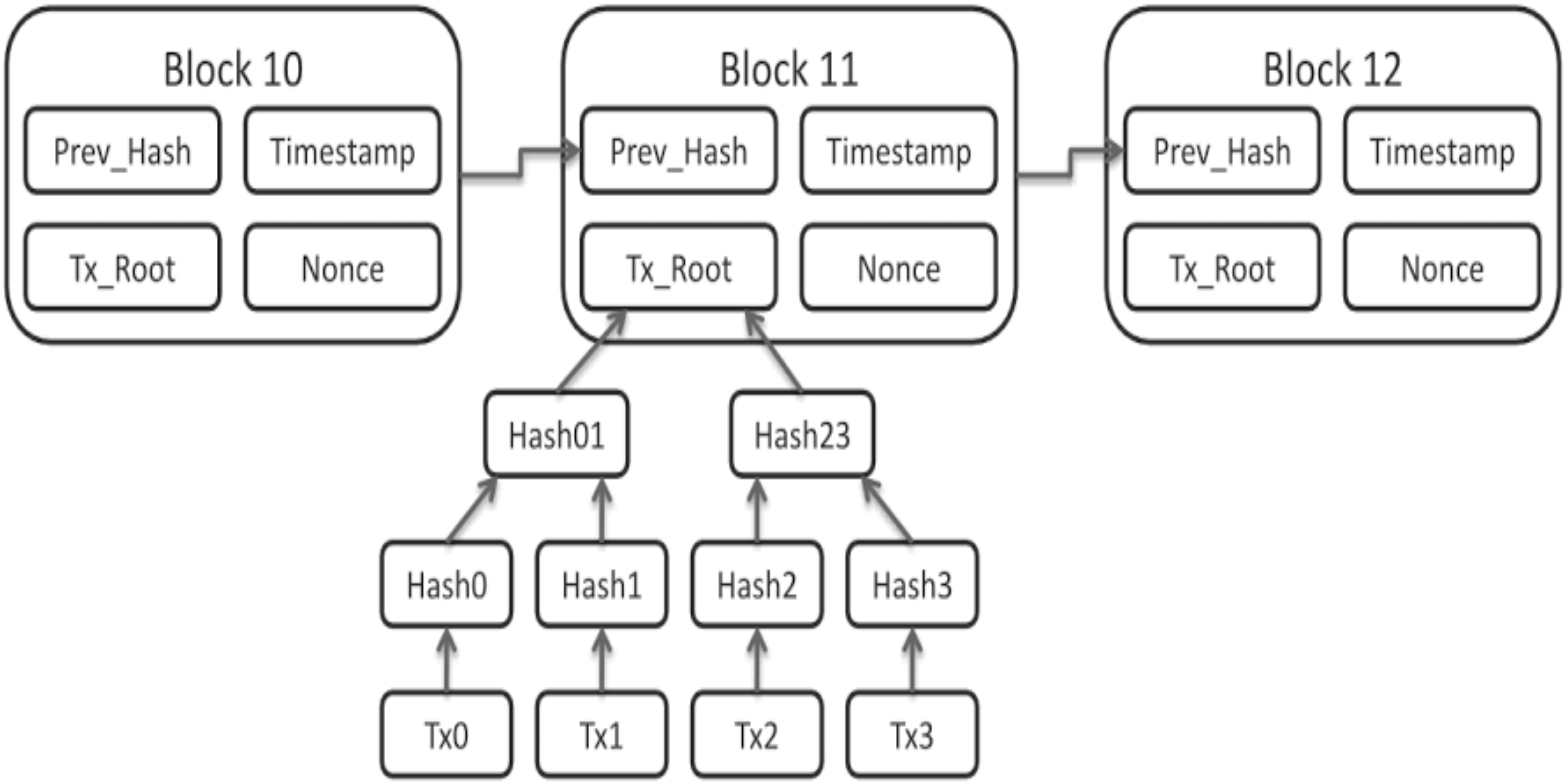


Triple-Entry Bookkeeping (Transaction-To-Transaction Payments) As Used By Bitcoin

Árvores de transação

- A ideia da árvore de transação é reunir todas as transações em uma árvore.
- Dessa forma, é possível sintetizar a informação de todas as transações calculando apenas o hash dela.
- Com a propriedade da injetividade do hash, é possível verificar que duas árvores são iguais em complexidade de tempo constante.

Árvore de transação



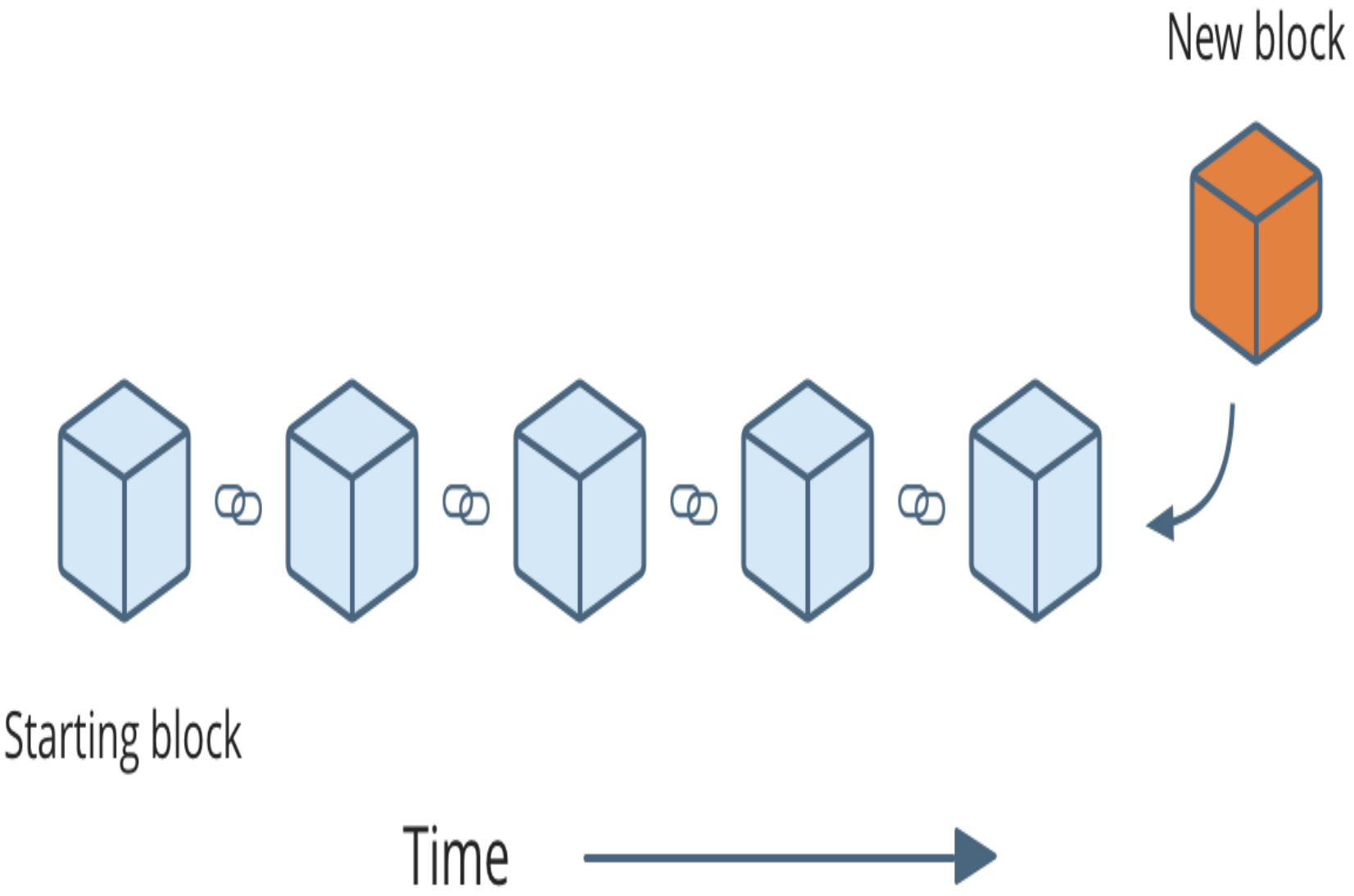
```
mutual
  data TXTree : (time : Time) (block : Nat)
    (outputs : List TXFieldWithId)
    (totalFees : Amount)
    (qtTransactions : tQtTxs) → Set where

    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)
```

```
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
```

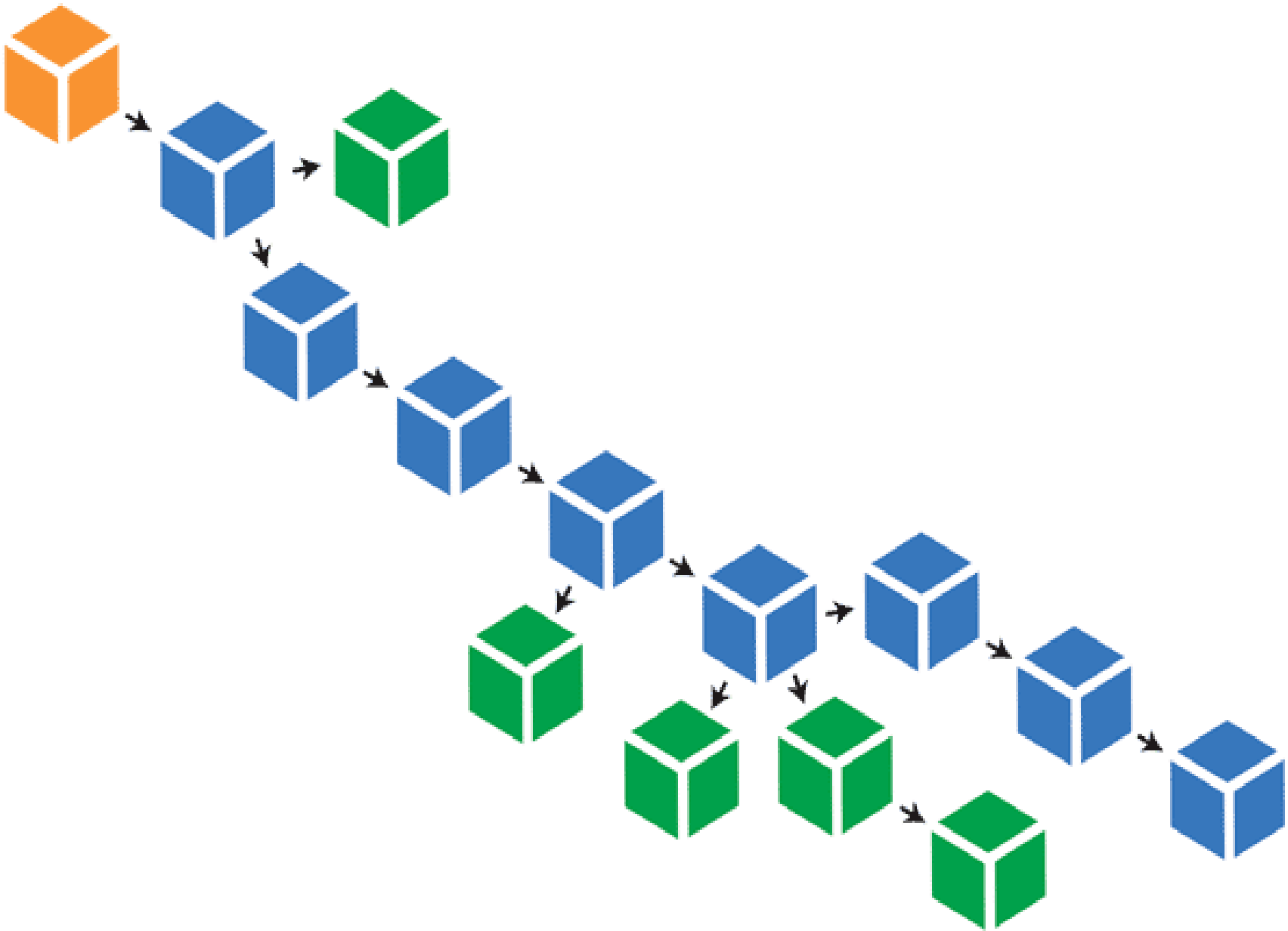
Blocos e cadeia de blocos

- As transações são incorporadas em blocos aonde são armazenadas.
- No Bitcoin, existe uma cadeia de bloco chamada de blockchain. O primeiro bloco é chamado de *genesis block* e a cada 10 minutos em média, um novo bloco é incorporado a essa cadeia de blocos.
- Existe um limite de transações que cada bloco pode possuir (no Bitcoin, é 10 MB de dados). Por isso, é importante pagar uma taxa de transação para que essa transação seja incluída na cadeia de blocos.
- Se existirem duas blockchains em computadores diferentes, considera-se a cadeia de blocos válida aquela que tiver mais blocos.

Blockchain



Blockchain



Mineração

- Todo bloco possui um problema criptográfico a ser resolvido. Ou seja, é preciso calcular um valor *nounce* tal que o hash do bloco seja menor que o um valor especificado.
- Esse valor especificado é calculado a partir da média dos últimos 2016 blocos (2 semanas) para que a média de mineração de blocos seja de 10 minutos.

```
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
```

Ressalvas

- Nesse trabalho, todas as especificações foram colocadas nos tipos.
- Em outros trabalhos, a abordagem pode ser diferente. Ou seja, sem nenhuma informação nos tipos e todas as provas feitas de forma separada.
- Essa abordagem não foi escolhida, pois é muito mais simples de entender o código quando os tipos são mais expressivos.

Detalhes do que falta

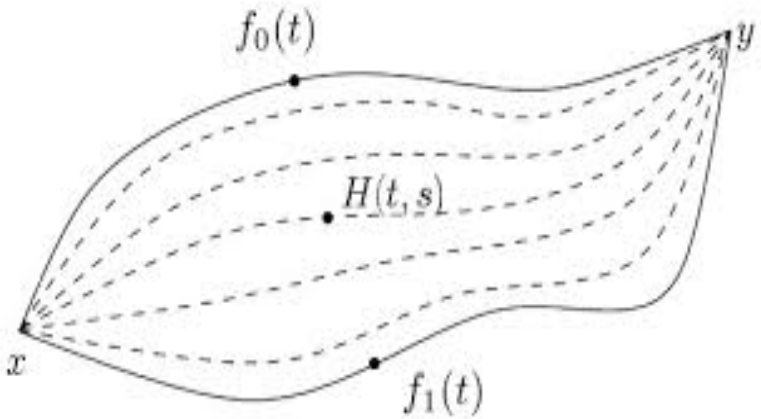
- Fazer o cálculo do hash do bloco e o campo nounce do bloco.
- Mudar o protocolo para receber o bloco inteiro ao invés de transações individuais.
- Adicionar o caso de existir uma blockchain de tamanho maior que a do que a do servidor.

Partes mais complexas de adicionar

- Fazer a modelagem formal da cadeia de blocos como um sistema distribuído.
- Criar código do cliente para interagir com os nós.
- Especificar o protocolo entre cliente e nó e entre nó com nó. Entretanto, a parte mais difícil já foi codificada, que é adicionar as provas ao dado puro que o cliente envia.
- Adicionar uma linguagem de script na criptomoeda.

Outras ideias

- Melhorar o sistema de tipos. Por enquanto, está se usando tipos dependentes, mas seria interessante utilizar tipos lineares para evitar o coletor de lixo ou usar *Cubical Type Theory* por ser mais expressivo.



Outras ideias

- Utilizar outros provadores de teoremas. O Lean 4 e CoQ possuem táticas, algo que Agda não possui. O Idris 2 possui tipos lineares.
- Utilizar teorias das categorias para deixar o código mais abstrato e conciso. Além de poder utilizar as bibliotecas já implementadas da linguagem.
- Utilizar bibliotecas para facilitar a transferência de mensagens entre nós diferentes. Cloud Haskell já facilita bastante esse processo.

Referências Bibliográficas

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

Norell, U. (2008). Dependently typed programming in agda. In *International school on advanced functional programming* (pp. 230–266).

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