Blockchain System

1 Definitions

Definition 1 (Set). *Let* **Elt** *be the set of (concrete) elements. Let* $\emptyset$ *be an empty set and* $\mathbf{e} \in \mathbf{Elt}$*. A set of elements is expressed as the following syntax:*
$\mathbf{s} ::= \emptyset \mid \mathbf{e} \mid \mathbf{s} :: \mathbf{s}$

Definition 2 (Account). *An account is expressed as a tuple* $<$ **als**, **pak**, **puk**, **pkh** $>$*, where* **als** *is the alias of the account,* **pak** *is its private key,* **puk** *is its public key and* **pkh** *is its public key hash .*

Definition 3 (Contract). *An contract is expressed as a tuple* $<$ **als**, **puh**, **code** $>$*, where* **als** *is the alias of the contract,* **puh** *is its public hash and* **code** *is the code of the contract.*

Definition 4 (Manager). *An manager manages accounts on the blockchain. It is expressed as a tuple* $<$ **puk**, **pkh**, **bal**, **cou** $>$*, where* **puk** *is the public key of an account,* **pkh** *is its public key hash,* **bal** *is its balance and* **cou** *is its counter whose form is a pair (*$\mathbf{n}$, $\mathbf{b}$*), where* $\mathbf{n}$ *is a nature number and* $\mathbf{b}$ *is a boolean value.*

Definition 5 (Contractor). *A contractor manages smart contracts on the blockchain. It is expressed as a tuple* $<$ **puh**, **bal**, **code**, **storage** $>$*, where* **puh** *is the public hash of the contract,* **bal** *is its balance,* **code** *is its code and* **storage** *is its storage.*

Definition 6 (Operation). *An operation is expressed as the following syntax:*
op *::= transfer* $\mathbf{n}$ *from* **pkh** *to* $\mathbf{pkh}'$ *fee* $\mathbf{m}$
| *originate contract* **id** *transferring* $\mathbf{n}$ *from* **pkh** *running* **code** *init* $\mathbf{s}$ *fee* $\mathbf{m}$
| *transfer* $\mathbf{n}$ *from* **pkh** *to* **puh** *arg* $\mathbf{s}$ *fee* $\mathbf{m}$

Definition 7 (Query). *An query is expressed as the following syntax:*
qry *::= get balance for* **pkh**
| *get status for* **oph**
| *get contract storage* **pkh**
| *get code for* **pkh**
| *get public key for* **pkh**
| *get counter for* **pkh**

Let $\mathbf{C}$ be the set of accounts, $\mathbf{O}$ be a set of operations, and $\mathbf{S}$ be the set of contracts.

Definition 8 (State of a node). *The state of a node is expressed as a tuple* [$\mathbf{C}$, $\mathbf{O}$, $\mathbf{S}$] .

When an operation is injected in a node, it enters in a pending pool (and called a pending operation).

Definition 9 (Pending operation). *A pending operation is expressed as a pair* $<$ **op**, **oph**, **t** $>$, *where* **op** *is an operation,* **oph** *is the operation hash and* **t** *is the time when it is injected.*

After sometime, a pending operation could be included in the blockchain as a accepted operation.

Definition 10 (Accepted operation). *An accepted operation is expressed as a tuple* $<$ **op**, **oph**, **t** $>$, *where* **op** *is an operation,* **oph** *is the operation hash and* **t** *is the time when it is included in the blockchain.*

Let **P** be a set of pending operations, **A** be a set of accepted operations, **K** be a set of managers, **T** be a set of contractors and **t** is the current time of the blockchain.

Definition 11 (Blockchain). *The state of a blockchain is expressed as a tuple* [**P**, **A**, **K**, **T**, **t**].

Definition 12 (Blockchain system). *A blockchain system* $\mathbf{S} \triangleq \langle\ \mathbf{M}, \mathbf{B}\ \rangle$ *consists of*

1. $\mathbf{M} \equiv$ *[C, O, S] is the state of a node, and*
2. $\mathbf{B} \equiv$ *[P, A, K, T, t] is the state of a blockchain such as* $\forall\ c \in C \implies \exists\ k \in K$, *k.pkh = c.pkh and* $\forall\ s \in S \implies \exists\ p \in T$, *s.puh = p.puh.*

2 Rules

2.1 Transfers

Rule 1 [proposal]:

$$\frac{\text{checkAcc}(pkh,\ \text{C})}{\langle[\text{C, O, S}],\ [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, (transfer } n \text{ from } pkh \text{ to } pkh' \text{ fee } m)\ ::\ \text{O, S}],\ [\text{P, A, K, T, t}]\rangle} \quad (1)$$

Rule 2 [injected]:

$$\frac{\text{checkBan(K, } pkh, n, m) \wedge \text{checkCou(K, } pkh) \wedge \text{checkPub(K, } pkh')}{\langle[\text{C, (transfer } n \text{ from } pkh \text{ to } pkh' \text{ fee } m) :: \text{O, S}],\ [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, O, S}],[(< \text{transfer } n \text{ from } puk \text{ to } puk' \text{ fee } m, \text{generateOph}(pkh, pkh', n, m, \text{t}), \text{t} >) :: \text{P, A, updateCou(K, } pkh, \text{True), T, t}]\rangle} \quad (2)$$

Rule 3 [rejected of counter]:

$$\frac{\neg\ \text{checkCou(K, } pkh\text{)}}{\langle[\text{C, (transfer } n \text{ from } pkh \text{ to } pkh' \text{ fee } m) :: \text{O, S}], [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, O, S}], [\text{P, A, K, T, t}]\rangle} \quad (3)$$

Rule 4 [rejected of balance]:

$$\frac{\neg\ \text{checkBan(K, } pkh\text{, } m\text{, } n\text{)}}{\langle[\text{C, (transfer } n \text{ from } pkh \text{ to } pkh' \text{ fee } m) :: \text{O, S}], [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, O, S}], [\text{P, A, K, T, t}]\rangle} \quad (4)$$

Rule 5 [rejected of public key]:

$$\frac{\neg\ \text{checkPub(K, } pkh'\text{)}}{\langle[\text{C, (transfer } n \text{ from } pkh \text{ to } pkh' \text{ fee } m) :: \text{O, S}], [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, O, S}], [\text{P, A, K, T, t}]\rangle} \quad (5)$$

Rule 6 [included]:

$$\frac{}{[< \text{transfer } n \text{ from } puk \text{ to } puk' \text{ fee } m, \text{ oph, t} > :: \text{P, A, K, T, t' }] \rightarrow [\text{P,}< \text{transfer } n \text{ from } puk \text{ to } puk' \text{ fee } m, \text{ oph, t' } > :: \text{A, updateSucc(K, } puk\text{, } puk'\text{, } n\text{, } m\text{), T, t' + 1}]} \quad (6)$$

Rule 7 [timeout]:

$$\frac{\text{t' - t} >= 60}{[< \text{transfer } n \text{ from } puk \text{ to } puk' \text{ fee } m, \text{ oph, t} > :: \text{P, A, K, T, t' }] \rightarrow [\text{P, A, updateCou(K, } puk\text{, False), T, t'}]} \quad (7)$$

2.2 Smart Contracts

A. Originate

Rule 1 [proposal]:

$$\frac{\text{checkAcc(}pkh\text{, C)} \wedge \text{checkId(}id\text{, S)} \wedge \text{checkPrg(}code\text{, }s\text{)}}{\langle[\text{C, O, S}], [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, (originate contract } puh \text{ transferring } n \text{ from } pkh \text{ running } code \text{ init } s) :: \text{O, S}], [\text{P, A, K, T, t}]\rangle} \quad (8)$$

Rule 2 [injected]:

$$\frac{\text{checkBan(K, } pkh\text{, } n\text{, } m) \wedge \text{checkCou(K, } pkh\text{)}}{\langle[\text{C, (originate contract } id \text{ transferring } n \text{ from } pkh \text{ running } code \text{ init } s)\\ :: \text{O, S], [P, A, K, T, t] } \rangle \rightarrow \langle\ [\text{C, O, S}],\\ [(\text{originate contract } id \text{ transferring } n \text{ from } pkh \text{ running } code \text{ init } s)\\ :: \text{P, A, updateCou(K, } pkh\text{, True), T, t] } \rangle} \quad (9)$$

Rule 3 [rejected of code]:

$$\frac{\neg\ \text{checkPrg(}code\text{, } s\text{)}}{\langle[\text{C, (originate contract } id \text{ transferring } n \text{ from } pkh \text{ running } code \text{ init } s)\\ :: \text{O, S], [P, A, K, T, t]}\rangle \rightarrow \langle[\text{C, O, S}], [\text{P, A, K, T, t}]\rangle} \quad (10)$$

Rule 4 [rejected of counter]:

$$\frac{\neg\ \text{checkCou(K, } pkh\text{)}}{\langle[\text{C, (originate contract } id \text{ transferring } n \text{ from } pkh \text{ running } code \text{ init } s))\\ :: \text{O, S], [P, A, K, T, t]}\rangle \rightarrow \langle\ [\text{C, O, S}], [\text{P, A, K, T, t}]\rangle} \quad (11)$$

Rule 5 [rejected of balance]:

$$\frac{\neg\ \text{checkBan(K, } pkh\text{, } n\text{, } m\text{)}}{\langle[\text{C, (originate contract } id \text{ transferring } n \text{ from } pkh \text{ running } code \text{ init } s)\\ :: \text{O, S], [P, A, K, T, t]}\rangle \rightarrow \langle[\text{C, O, S}], [\text{P, A, K, T, t}]\rangle} \quad (12)$$

Rule 6 [included]:

$$\frac{}{\langle[\text{C, O, S}], [< (\text{originate contract } id \text{ transferring } n \text{ from } pkh \text{ running}\\ code \text{ init } s), \text{t} > :: \text{P, A, K, T, t' }]\rangle \rightarrow \langle[\text{C, O, addContr(S, } id\text{,}\\ \text{generateHash(}id\text{, } code\text{, } s\text{, t'), } code\text{)}], [\text{P,}< (\text{originate contract } id\\ \text{transferring } n \text{ from } pkh \text{ running } code \text{ init } s), \text{t' } > :: \text{A, updateSucc(K,}\\ puk\text{, } n\text{, } m\text{), addOrig(T, } < \text{generateHash(}id\text{, } code\text{, } s\text{, t'), 0, } code\text{,}\\ \text{getStorage(}code\text{, } s\text{) } >), \text{t' + 1}]\rangle} \quad (13)$$

Rule 7 [timeout]:

$$\frac{\text{t' - t >= 60}}{[< (\text{originate contract } id \text{ transferring } n \text{ from } pkh \text{ running } code \text{ init } s) >\\ :: \text{P, A, K, T, t' }] \rightarrow [\text{P, A, updateCou(K, } puk\text{, False), T, t']}} \quad (14)$$

B. Transfer

Rule 1 [proposal]:

$$\frac{\text{checkAcc}(pkh,\ \text{C})}{\langle[\text{C, O, S}],\ [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, (transfer } n \text{ from } pkh \text{ to } puh \text{ arg } s \text{ fee } m) :: \text{O, S}],\ [\text{P, A, K, T, t}]\rangle} \tag{15}$$

Rule 2 [injected]:

$$\frac{\text{checkBan(K, } pkh, n, m) \wedge \text{checkCou(K, } pkh) \wedge \text{checkContr(T, } puh, \text{s})}{\langle[\text{C, (transfer } n \text{ from } pkh \text{ to } puh \text{ arg } s \text{ fee } m) :: \text{O, S}],\ [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, O, S}],\ [(< (\text{transfer } n \text{ from } pkh \text{ to } puh \text{ arg } s \text{ fee } m), \text{generateOph}(pkh, puh, s, n, m, \text{t} >) :: \text{P, A, updateCou(K, } pkh, \text{True}), \text{T, t}]\ \rangle} \tag{16}$$

Rule 3 [rejected of counter]:

$$\frac{\neg\ \text{checkCou(K, } pkh)}{\langle[\text{C, (transfer } n \text{ from } pkh \text{ to } puh \text{ arg } s \text{ fee } m) :: \text{O, S}],\ [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, O, S}],\ [\text{P, A, K, T, t}]\rangle} \tag{17}$$

Rule 4 [rejected of balance]:

$$\frac{\neg\ \text{checkBan(K, } pkh, n, m)}{\langle[\text{C, (transfer } n \text{ from } pkh \text{ to } puh \text{ arg } s \text{ fee } m) :: \text{O, S}],\ [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, O, S}],\ [\text{P, A, K, T, t}]\rangle} \tag{18}$$

Rule 5 [rejected of public key]:

$$\frac{\neg\ \text{checkContr(T, } puh)}{\langle[\text{C, (transfer } n \text{ from } pkh \text{ to } puh \text{ arg } s \text{ fee } m) :: \text{O, S}],\ [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, O, S}],\ [\text{P, A, K, T, t}]\rangle} \tag{19}$$

Rule 6 [rejected of argument]:

$$\frac{\neg\ \text{checkArg(T, } puh\text{, } s)}{\langle[\text{C, (transfer } n \text{ from } pkh \text{ to } puh \text{ arg } s \text{ fee } m) :: \text{O, S}], [\text{P, A, K, T, t}]\rangle \rightarrow \langle[\text{C, O, S}], [\text{P, A, K, T, t}]\rangle} \quad (20)$$

Rule 7 [included]:

$$\frac{}{[< \text{transfer } n \text{ from } puk \text{ to } puh \text{ fee } m, \text{ oph, t} > :: \text{P, A, K, T, t' }] \rightarrow [\text{P, } < (\text{transfer } n \text{ from } pkh \text{ to } puh \text{ arg } s \text{ fee } m), \text{ oph, t' } > :: \text{A, updateSucc(K, puk, ´´, } n\text{, } m), \text{ updateConstr(T, } puh\text{, } s), \text{ t' + 1}]} \quad (21)$$

Rule 8 [timeout]:

$$\frac{\text{t' - t} >= 60}{[\text{transfer } n \text{ from } puk \text{ to } puh \text{ fee } m, \text{ t} > :: \text{P, A, K, T, t' }] \rightarrow [\text{P, A, updatecou(K, } puk\text{, False), T, t'}]} \quad (22)$$

3 Functions

1. Function `checkAcc(pkh, C)` checks whether an account *pkh* exists in C
2. Function `checkPub(K, pkh)` checks whether the public key of the public key hash *pkh* is reveled to the blockchain.
3. Function `checkBan(K, pkh, n, m)` checks whether the balance of the account *pkh* is greater or equal to $m + n$
4. Function `checkCou(K, pkh)` checks whether the current counter of an account *phk* is used
5. Function `updateSuc(K, pkh, pkh', n, m)` updates the balance and the counter of the account *phk* and the balance of the account phk', where
   - `< puk, pkh, bal, (n, True) > =>`
     `< puk, pkh, bal - n - m, (n + 1, False) >`
   - `< puk', pkh', bal', cou' > => < puk', pkh', bal' + n, cou' >`
6. Function `updateCou(K, puk, b')` updates the counter of the account *phk*, where
   - `< puk, pkh, bal, (n, b) > => < puk, pkh, bal, (n, b') >`
7. Function `checkId(id, S)` checks whether a contract *id* exists in S
8. Function `checkPrg(code, s)` checks whether the code *code* are well type and *s* is well type input
9. Function `addContr(S, id, puh, code)` adds a new contract $<$ *id*, *puh*, *code* $>$ into S
10. Function `generateOph(pkh,pkh, n, m, t)` generates a operation hash
11. Function `generateHash(S, id, puh, code, t)` generates the public hash of a contract
12. Function `addOrig(T,< hash, 0, code, storage >)` add the a new originator $<$ *puh*, 0, *code*, *storage* $>$
13. Function `getStorage(code, s)` gets the storage for the code *code* and the input *s*

4 Some implementations

Function `checkAcc(puh, C)` checks whether an account exists and and `checkPuk(puh, K)` checks the revelation of its public key to the blockchain.

```
let rec checkAcc puh C =
  match C with
  | 0 -> false
  | < als, pak, puk, pkh' > :: C' ->
    if (puh = puh')  then true
    else checkAcc (puh, C')

let rec checkPuk puh K =
  match C with
  | 0 -> false
  | < als, pak, puk, pkh' > :: K' ->
    if (puh = puh') and (puk =/= nil) then true
    else 5checkPuk (puh, K')
```

The following functions interact with **K**.

```
let rec checkBal K puk n m =
  match K with
  | 0 -> true
  | < puk', bal, cou > :: K' ->
    if (puk = puk') and (n + m) <= bal then true
    else checkBal (K', puk, n, m)

let rec checkPub K puk =
  match K with
  | 0 -> false
  | < puk', bal, cou > :: K' ->
    if (puk = puk') then true
    else checkExi (K', puk)

let rec checkCou K puk =
  match K with
  | 0 -> false
  | < puk', bal, cou > :: K' ->
    if (puk = puk') and (cou = T) then true
    else checkCou (K', puk)

let rec updateCou K puk =
  match K with
  | 0 -> 0
  | < puk', bal, cou > :: K' ->
    if (puk = puk') then < puk', bal, F > :: K'
    else < puk', bal, cou > :: updateCou (K', puk)
```

```
let rec updateSuc K puk puk' m n =
  match K with
  | 0 -> 0
  | < puk'', bal, cou > :: K' ->
    if (puk = puk'') then < puk'', bal - (n + m), T >
       :: updateSuc (K', puk, puk', n, m)
    else if (puk' = puk'') then < puk'', bal + n, cou > :: K'
         else  < puk'', bal, cou >
               ::updateSuc (K', puk, puk', n, m)
```