

A Typed Programmatic Interface to Contracts on the Blockchain

Thi Thu Ha Doan^[0000–0001–7524–4497] and Peter Thiemann^[0000–0002–9000–1239]

University of Freiburg, Germany
`{doanha,thiemann}@informatik.uni-freiburg.de`

Abstract. Smart contract applications on the blockchain can only reach their full potential if they integrate seamlessly with traditional software systems via a programmatic interface. This interface should provide for originating and invoking contracts as well as observing the state of the blockchain. We propose a typed API for this purpose and establish some properties of the combined system. Specifically, we provide an execution model that enables us to prove type-safe interaction between programs and the blockchain. We establish further properties of the model that give rise to requirements on the API. A prototype of the interface is implemented in OCaml for the Tezos blockchain.

Keywords: smart contracts · embedded domain specific languages · types.

1 Introduction

First generation blockchains were primarily geared towards supporting cryptocurrencies. Bitcoin is the most prominent system of this kind [14]. Although Bitcoin already features a rudimentary programming language called Script, second generation blockchains like Ethereum [5] feature Turing-complete programming facilities, called *smart contracts*. They provide robust trustworthy distributed computing facilities even though the programs run on a peer-to-peer network with untrusted participants. Each peer in the network runs the same program and uses cryptographic methods to check the results among the other peers and to create a persistent ledger of all transactions, the blockchain, thus ensuring the integrity of the results. Third generation blockchains, like Tezos [8], are adaptable to new requirements without breaking participating peers (no “soft forks” required and “hard forks” can be avoided).

The strength of programs on the blockchain is also their weakness. They are fully deterministic in that they can only depend on data that is ultimately stored on the chain including the parameters of a contract invocation. Moreover, the code, the data, as well as all transactions are public. These properties make it hard to react to external stimuli like time triggers or events like a price exceeding a threshold unless these stimuli get translated to contract invocations.

Arguably, smart contracts are more useful if they can be integrated with traditional software systems and thus triggered from outside the blockchain.

Oracles [13,6] provide an approach for contracts to obtain outside information. A contract registers a request and a callback with an oracle. The oracle invokes the callback as soon as an answer is available.

There are other usecases for connecting a contract with traditional software. One example is automating procedures like managing an auction, bidding in an auction, optimizing fees, or initiating delivery of goods to a customer. While some of these procedures are amenable to implementation as contracts, we might want to save the fee of running them on the blockchain. In particular, for actions that happen strictly within a single domain of trust, it is not worth running them on the blockchain. For example, automated bidding runs on behalf of a single peer.

Building such automation requires a programmatic interface to implement the interactions. Current blockchains mostly provide RPC interfaces, such as the Ethereum JSON-RPC API [7] and the Tezos RPC API [8], but they require cumbersome manipulation of string data in JSON format and do not provide static guarantees (except that the response to a well-formed JSON input is also a well-formed JSON output). To improve on this situation we present a typed API for invoking contracts from OCaml programs. Our typed API supports the implementation of application programs and oracles that safely interact with smart contracts on the blockchain. Moreover, our approach provides a type-safe facility to communicate with contracts where data is automatically marshalled between OCaml and the blockchain. This interface is a step towards a seamless integration of contracts into traditional programs.

Contributions

- A typed API for originating and invoking contracts as well as querying the state of the blockchain.
- An operational semantics for functional programs running alongside smart contracts in a blockchain.
- Established various properties of the combined system with proofs in upcoming techreport.
- An implementation of a low-level OCaml-API to the Tezos blockchain, which corresponds to the operational semantics.¹

There is an extended version of the paper with further proofs.²

2 Motivation

Suppose you want to implement a bidding strategy for an auction that is deployed on the blockchain as a smart contract. Your bidding strategy may start at a certain amount and increase the bid until a limit is reached. Of course, you

¹ Available at <https://github.com/tezos-project/Tezos-OCaml-API>.

² Available at <https://arxiv.org/abs/2108.11867>.

```

parameter (or (unit %close)
              (unit %bid)); # bid in transfer
storage (pair bool # bidding allowed
        (pair address # contract owner
          address # highest bidder's address
        ));

```

Listing 1.1. Header of the auction contract

```

# let auction = Cl.make_contract_hash auction_hash
#   ~parameter:(Ct.Or (Ct.Unit, Ct.Unit))
#   ~storage:(Ct.Pair (Ct.Bool, Ct.Pair (Ct.Addr, Ct.Addr))));
val auction :
  ((unit, unit) Either.t,
   bool * (Cl.Addr.t * Cl.Addr.t)) Cl.contract

```

Listing 1.2. Getting the auction handle

only want to increase your bid if someone else placed a higher bid. So you want to write a program to implement this strategy.

This task cannot be implemented as a smart contract without cooperation of the auction contract because it reacts on external triggers. Bidding requires watching the current highest bid of the contract and react if another bidder places a higher bid. The auction contract could anticipate the need for such observations by allowing bidders to register callbacks that are invoked when a higher bid arrives. However, we cannot assume such cooperation of the auction contract nor would we be willing to pay the the fee for running that callback.

For concreteness, Listing 1.1 shows the header of an auction contract in Michelson [12]. The parameter clause specifies the contract’s parameter type. It is a sum type (indicated by `or`) and each alternative constitutes an entrypoint, named `%close` and `%bid`. The caller selects the entrypoint by injecting the argument into the left or right summand. Both entrypoints take a unit parameter. The `%bid` entrypoint considers the transferred tokens as the bid. The storage clause declares the state of the contract, which is a nested pair type indicating whether bidding is allowed (`bool`), the address of the contract owner (to prohibit unauthorized calls to `%close`), and the bidder’s address. The highest bid corresponds to the token balance of the contract.

We only outline the implementation of the entrypoints. The `%close` entrypoint first checks its sender’s address against the owner’s address in the store. Then it transfers the funds to the owner, closes the contract by clearing the bidding flag, and leaves it to the owner to deliver the goods.³ The `%bid` entrypoint immediately returns each bid that is not higher than the existing highest bid. Otherwise, it keeps the funds transferred, returns the previous highest bid to its owner, and stores the current bidder as the new highest bidder.

We present a program that implements strategic bidding by interacting with the blockchain. The bidding strategy cannot be implemented as a smart contract.

³ For simplicity we elide safeguarding by a third-party oracle.

```

let rec poll limit step =
  let (bidding, (_, highest_bidder)) = Cl.get_storage auction;
  let high_bid = Cl.get_balance auction;
  if bidding && high_bid < limit then
    (if highest_bidder <> my_address then (* entrypoint %bid *)
     try
       Cl.call_contract auction
         (right (min (high_bid + step, limit)))
     with
       | Cl.FAILWITH message -> poll limit step;
    Time.sleep(5 * 60);
  poll limit step)

```

Listing 1.3. Bidding strategy

In Listing 1.2, we use the library function `Cl.make_contract_from_hash` to obtain a typed handle for the contract.⁴ The function takes the hash of the contract along with representations of the types of the parameter and the storage (from module `Ct`). It checks the validity of the hash and the types with the blockchain and returns a typed handle, which is indexed with OCaml types corresponding to parameter and storage type.

The implementation of the bidding strategy in Listing 1.3 first checks the state of the contract to find the current highest bid. As long as bidding is allowed and the current bid is below our limit, we update our bid by a given amount `step`, and then keep watching the state of the contract by polling it every five minutes.

The functions `get_storage` and `get_balance` obtains the storage and current balance, respectively, of a contract from the blockchain. They never fail. Function `call_contract` takes a typed handle and a parameter of suitable type. It indicates failure by raising an exception. If failure is caused by the `FAILWITH` instruction in the contract, then the corresponding `Cl.FAILWITH` exception is raised, which carries a string corresponding to the argument of the instruction. In our particular example, the auction may fail with signaling the message “closed” or “bid_too_low”. Our code ignores this message for simplicity.

This code is idealized in several respects. Originating or running a contract requires proposing a fee to the blockchain, which may or may not be accepted. Starting a contract may also time out for a variety of reasons. So just invoking a contract with a fixed fee does not guarantee the contract’s execution. Even if the invocation is locally accepted, it still takes a couple of cycles before we can be sure the invocation is globally accepted and incorporated in the blockchain. Hence, after starting the invocation, we have to observe the fate of this invocation. If it does not get incorporated, then we need to analyze the reason and react accordingly. For example, if the invocation was rejected because of an insufficient fee, we might want to restart with an increased fee. Or we might decide to wait until the invocation goes through without increasing the fee.

⁴ `Cl` is the module containing the contract library.

Hence, we would implement a scheme similar to the bidding strategy: start with a low fee and increase (or wait) until the contract is accepted or a fee cap is reached. On the other hand, an observer function like `get_state` always succeeds.

The low-level interface that we propose in this paper requires the programmer to be explicit about fees, waiting, and polling the state of contract invocations.

In summary, a useful smart-contract-API has facilities to

- query the current state of the blockchain (e.g., fees in the current block),
- query storage and balance of a contract (to obtain the current highest bid),
- originate contracts, invoke contracts, and initiate transfers. Hence, the API has to run on behalf of some account (by holding its private key).

These facilities are supported by the (untyped) RPC interface of the Tezos blockchain, which is the basis of our implementation.

3 Execution Model

The context of our work is the Tezos blockchain [8,2]. Tezos is a self-amending blockchain that improves several aspects compared to established blockchains. Tezos proposes an original consensus algorithm, Liquid Proof of Stake, that applies not only to the state of its ledger, like Bitcoin [14] or Ethereum [5], but also to upgrades of the protocol and the software.

Tezos supports two types of accounts: implicit accounts, which are associated with a pair of private/public keys, and smart contracts, which are programmable accounts created by an origination operation. The address of a smart contract is a unique public hash that depends on the creation operation. No key pair is associated with a smart contract. An implicit account is maintained on the blockchain with its public key and balance. A smart contract account is stored with its script, storage, and balance. A contract script maps a pair of a parameter and a storage, which have fixed and monomorphic types, to a pair of a list of internal operations and an updated storage. An account can perform three kinds of transactions: (1) transfer tokens to an implicit account, (2) invoke a smart contract, or (3) originate a new smart contract. A contract origination specifies the script of the contract and the initial contents of the contract storage, while a contract invocation must provide input data. Each transaction contains a fee to be paid either by payment to a baker or by destruction (burning). A transaction is injected into the blockchain network via a node, which then validates the transaction before submitting it to the network. A transaction may be rejected by the node for a number of reasons. After validation, the transaction is injected into a *mempool*, which contains all pending transactions before they can be included in a block. A pending transaction may simply disappear from the mempool, for example, a transaction times out when 60 blocks have passed and it can no longer be included in a block. When a transaction is included in the blockchain, the affected accounts are updated according to the transaction result.

The execution model consists of functional (OCaml) programs that interact with an abstraction of the Tezos blockchain [8]. As the blockchain is realized by

a peer-to-peer network of independent nodes, interaction happens through *local nodes* that receive requests to originate and invoke contracts from programs that run on a particular node. We model the blockchain itself as a separate, abstract global entity that represents the current consensual state of the system. Our model does not express low-level details, but relies on nondeterminism to describe the possible behaviors of the system. In particular, we do **not** formalize the execution of the smart contracts themselves, we rather consider them as black boxes and probe their observable behavior. Tezos’s smart contract language Michelson and its properties have been formalized elsewhere [4].

We write \emptyset for the empty set and $\mathbf{e} :: \mathbf{s}$ to decompose a set nondeterministically into an element \mathbf{e} and a set \mathbf{s} . We generally use lowercase boldface for metavariables ranging over values of a certain syntactic category, e.g., **puk** for public keys, and the capitalized name for the corresponding type as well as for the set of these values (as in Puk).

3.1 Local Node

A local node runs on behalf of authorities, which are called *accounts* in Tezos. An account is represented by a key pair $\langle \mathbf{pak}, \mathbf{puk} \rangle$, where **pak** is a private key and **puk** the corresponding public key in a public key encryption scheme.

The local node offers operations to transfer tokens from one account to another, to invoke a contract, and to originate a contract on the blockchain.

op ::= transfer **nt** from **puk** to **addr** arg **p** fee **fee**
 | originate contract transferring **nt** from **puk** running **code** init **s** fee **fee**

In the transfer, which also serves as contract invocation, **nt** is the amount of tokens transferred, **puk** is the public key of the sender, **addr** is either a public key for an implicit account (in case of a simple transfer) or a public hash for a smart contract (for an invocation), **p** is the argument passed to the smart contract, which is empty for a simple transfer, and **fee** is the amount of tokens for the transaction fee. In originate, **code** is the script of a smart contract and **s** is the initial value of the contract’s storage. Each operation returns an *operation hash* **oph**, on which we can query the status of the operation.

The local node offers several ways to query the current state of the blockchain. Some *query operators* are defined by the following grammar:

qop ::= balance | status | storage | contract | ...

We obtain the balance associated with an implicit account or a contract by its public key or public hash, respectively; the status of a submitted operation by its operation hash; the stored value of a contract by its public hash; and the public hash of a contract by the operation hash of its originating transaction.

The domain-specific types come with different guarantees. Values of type Puh and Puk as well as Addr are not necessarily valid, as there might be no contract associated with a hash / no account associated with a public key. In contrast,

$$\begin{aligned}
\mathbf{c} &::= \mathbf{i} \mid \text{fix} \mid \mathbf{oph} \mid \mathbf{puh} \mid \mathbf{puk} \mid \mathbf{code} \mid \mathbf{nt} \mid () \mid \text{False} \mid \text{True} \\
\mathbf{st} &::= \text{pending} \mid \text{included}(\mathbf{i}) \mid \text{timeout} \\
\mathbf{err} &::= \mathbf{xPrg} \mid \mathbf{xBal} \mid \mathbf{xCount} \mid \mathbf{xFee} \mid \mathbf{xPub} \mid \mathbf{xPuh} \mid \mathbf{xArg} \mid \mathbf{xInit} \\
\mathbf{e} &::= \mathbf{c} \mid \mathbf{st} \mid \mathbf{err} \mid \mathbf{x} \mid \lambda \mathbf{x}. \mathbf{e} \mid \mathbf{ee} \mid \mathbf{e} + \mathbf{e} \mid \mathbf{e} = \mathbf{e} \mid \mathbf{e} \text{ and } \mathbf{e} \mid \mathbf{e} \text{ or } \mathbf{e} \mid \text{not } \mathbf{e} \\
&\quad \mid (\mathbf{e}, \mathbf{e}) \mid \text{nil} \mid \text{cons } \mathbf{e} \mathbf{e} \mid \text{left } \mathbf{e} \mid \text{right } \mathbf{e} \mid \text{some } \mathbf{e} \mid \text{none} \mid \text{match } \mathbf{e} \text{ with } \mathbf{pat} \rightarrow \mathbf{e} \dots \\
&\quad \mid \text{raise } \mathbf{e} \mid \text{try } \mathbf{e} \text{ except } \mathbf{e} \mid (\mathbf{e} : T \Rightarrow U) \\
&\quad \mid \mathbf{qop } \mathbf{e} \mid \text{transfer } \mathbf{e} \text{ from } \mathbf{e} \text{ to } \mathbf{e} \text{ arg } \mathbf{e} \text{ fee } \mathbf{e} \\
&\quad \mid \text{originate contract transferring } \mathbf{e} \text{ from } \mathbf{e} \text{ running } \mathbf{e} \text{ init } \mathbf{e} \text{ fee } \mathbf{e} \\
\mathbf{pat} &::= \mathbf{x} \mid (\mathbf{pat}, \mathbf{pat}) \mid \text{nil} \mid \text{cons } \mathbf{pat} \mathbf{pat} \mid \text{left } \mathbf{pat} \mid \text{right } \mathbf{pat} \mid \text{some } \mathbf{pat} \mid \text{none} \\
&\quad \mid \text{False} \mid \text{True} \mid \mathbf{st} \mid \mathbf{err} \\
T, U &::= \text{Puh} \mid \text{Puk} \mid \text{Addr} \mid \text{Cont } T \ U \mid \text{Code } T \ U \mid \text{Oph } T \ U \mid \text{Status} \mid \text{Exc} \mid \text{Tz} \\
&\quad \mid \top \mid \text{Int} \mid \text{Unit} \mid \text{Bool} \mid \text{Str} \mid T \rightarrow U \mid \text{Pair } T \ U \mid \text{List } T \mid \text{Or } T \ U \mid \text{Option } T
\end{aligned}$$

Fig. 1. Syntax of expressions, \mathbf{e} , and types, T

a value of type $\text{Cont } T$ is a public hash that is verified to be associated with a contract with parameter type T . Operation hashes \mathbf{oph} are only returned from blockchain operations. As the surface language neither contains literals of type Oph nor are there casts into that type, all values of Oph are valid.

Definition 1. *The state of a node is a pair $\mathbf{N} = [\bar{\mathbf{e}}, \mathbf{A}]$, where $\bar{\mathbf{e}}$ is a set of programs and $\mathbf{A} \subseteq \text{Pak} \times \text{Puk}$ is a set of implicit accounts.*

Queries and operations are started by closed expressions of type unit that run on the local node. Each program can send transactions on behalf of any account on the local node. Figure 1 defines the syntax of lambda calculus with sum, product, list, and option types, exceptions and fixpoint. Pattern matching is the only means to decompose values, cf. \mathbf{pat} . The execution model envisions off-chain programs interacting with smart contracts on the blockchain. The programs are defined using expression scripts. The off-chain scripts run on behalf of a single entity.

Domain-specific primitive types and constants \mathbf{c} support blockchain interaction, as well as several exceptional values collected in \mathbf{err} . There is syntax to initiate transfers and to originate contracts as well as for the queries. Finally, there is a type cast $(\mathbf{e} : T \Rightarrow U)$, which we describe after discussing types. An implementation provides all of these types and operations via a library API.

Types (also in Figure 1) comprise some base types as well as functions, pairs, lists, sums, and option types. These types are chosen to match with built-in types of Michelson. There are domain specific types of public hashes Puh and public keys Puk subsumed by a type of addresses Addr . $\text{Cont } T \ U$ is the type of a contract with parameter type T and storage type U . $\text{Code } T \ U$ indicates a Michelson program with parameter type T and storage type U . Tezos tokens have type Tz .

$$\begin{aligned}
\mathbf{E} &::= [] \mid \mathbf{sc}[\bar{\mathbf{v}} \ \mathbf{E} \ \bar{\mathbf{e}}] \mid \text{raise } \mathbf{E} \mid \text{try } \mathbf{E} \text{ except } \mathbf{e} \mid \text{match } \mathbf{E} \text{ with } \mathbf{pat} \rightarrow \mathbf{e} \dots \\
\mathbf{v} &::= \mathbf{c} \mid \mathbf{st} \mid \mathbf{err} \mid \lambda x. \mathbf{e} \mid (\mathbf{v}, \mathbf{v}) \mid \mathbf{nil} \mid \text{cons } \mathbf{v} \ \mathbf{v} \mid \text{left } \mathbf{v} \mid \text{right } \mathbf{v} \mid \text{some } \mathbf{v} \mid \mathbf{none}
\end{aligned}$$
Fig. 2. Evaluation contexts and values

The type $\text{Oph } T \ U$ signifies operation hashes returned by blockchain operations. The parameters of the hash carry the types when originating a contract. Otherwise, they are set to the irrelevant type \top . We take the liberty of omitting irrelevant type parameters, that is, we write Oph for $\text{Oph } \top \ \top$. Querying the status of an operation returns a value of type Status . Exceptions have type Exc .

Figure 2 defines evaluation contexts \mathbf{EC} and values \mathbf{v} . Here \mathbf{sc} ranges over the remaining syntactic constructors, which are treated uniformly: evaluation proceeds from left to right. Values are standard for call-by-value lambda calculus.

Type casts are only applicable to certain pairs of types governed by a relation $<:$, which could also serve as a subtyping relation. It is given by the axioms $\text{Puh } <: \text{Addr}$, $\text{Puk } <: \text{Addr}$, and $\text{Cont } T \ U <: \text{Puh}$. A cast from T to U is only allowed if $T <: U$ (upcast) or $U <: T$ (downcast). Upcasts always succeed, but downcasts may fail at run time. In particular, public hashes and public keys can both stand for addresses. Moreover, a smart contract with parameter type T is represented by its public hash at run time. The corresponding downcast must check whether the public hash is valid and has the expected parameter and storage type.

Figure 3 presents selected typing rules for expressions. We rely on an external typing judgment $\vdash_C \mathbf{code} : T$ for the contract language, which we leave unspecified, and $\vdash_V \mathbf{s} : T$ for serialized values as stored on the blockchain. The latter judgment states \mathbf{s} is a string parseable as a value of type T .

3.2 Global Structures

Our execution model abstracts from the particulars of the blockchain implementation, like the peer-to-peer structure or the distributed consensus protocol. Hence, we represent the blockchain by a few global entities: managers, contractors, and a pool of operations.

A *manager* keeps track of a single implicit account. Managers are represented by a partial map $\mathbf{M} : \text{Puk} \hookrightarrow \text{Bal} \times \text{Cnt}$. If $\mathbf{M}(\mathbf{puk}) = \langle \mathbf{bal}, \mathbf{cnt} \rangle$ is defined, then \mathbf{puk} is the public key of an account, \mathbf{bal} is its balance and \mathbf{cnt} is its counter whose form is a value-flag pair $(n, b) \in \mathbf{N} \times \text{Bool}$, where n is the value of the counter and b is its flag. The counter is used internally to serialize transactions.

A *contractor* manages a single smart contract. Contractors are represented by a partial map $\mathbf{C} : \text{Puh} \hookrightarrow \text{Code} \times \mathbf{t} \times \text{Bal} \times \text{Storage}$. If $\mathbf{C}(\mathbf{puh}) = \langle \mathbf{code}, \mathbf{t}, \mathbf{bal}, \mathbf{storage} \rangle$ is defined, then \mathbf{puh} is the public hash of a contract, \mathbf{code} is its code, \mathbf{t} is the time when it was accepted, \mathbf{bal} is its current balance, and $\mathbf{storage}$ is its current storage. The hash \mathbf{puh} is self-verifying as it is calculated from the fixed components \mathbf{code} and \mathbf{t} . All time stamps will be different in our model.

$$\begin{array}{c}
\Gamma \vdash \mathbf{i} : \text{Int} \quad \Gamma \vdash \mathbf{oph} : \text{Oph } T \ U \quad \Gamma \vdash \mathbf{puh} : \text{Puh} \quad \Gamma \vdash \mathbf{puk} : \text{Puk} \\
\\
\frac{\vdash_C \mathbf{code} : \text{Pair } T_p \ T_s}{\Gamma \vdash \mathbf{code} : \text{Code } T_p \ T_s} \quad \Gamma \vdash \mathbf{nt} : \text{Tz} \quad \Gamma \vdash () : \text{Unit} \quad \Gamma \vdash \text{False} : \text{Bool} \\
\\
\Gamma \vdash \text{True} : \text{Bool} \quad \Gamma \vdash \text{pending} : \text{Status} \quad \Gamma \vdash \text{timeout} : \text{Status} \\
\\
\frac{\Gamma \vdash \mathbf{e} : \text{Int}}{\Gamma \vdash \text{included}(\mathbf{e}) : \text{Status}} \quad \Gamma \vdash \mathbf{err} : \text{Exc} \quad \Gamma \vdash \mathbf{x} : \Gamma(\mathbf{x}) \quad \frac{\Gamma, \mathbf{x} : T' \vdash \mathbf{e} : T}{\Gamma \vdash \lambda \mathbf{x}. \mathbf{e} : T' \rightarrow T} \\
\\
\frac{\Gamma \vdash \mathbf{e} : T' \rightarrow T \quad \Gamma \vdash \mathbf{e}' : T'}{\Gamma \vdash \mathbf{e} \ \mathbf{e}' : T} \quad \frac{\Gamma \vdash \mathbf{e} : T \quad \Gamma \vdash \mathbf{e}' : T'}{\Gamma \vdash (\mathbf{e}, \mathbf{e}') : \text{Pair } T \ T'} \quad \frac{\Gamma \vdash \mathbf{e} : \text{Exc}}{\Gamma \vdash \text{raise } \mathbf{e} : T} \\
\\
\frac{\Gamma \vdash \mathbf{e} : T \quad \Gamma \vdash \mathbf{e}' : \text{Exc} \rightarrow T}{\Gamma \vdash \text{try } \mathbf{e} \ \text{except } \mathbf{e}' : T} \quad \frac{\Gamma \vdash \mathbf{e} : T \quad T <: U \vee U <: T}{\Gamma \vdash (\mathbf{e} : T \Rightarrow U) : U}
\end{array}$$

Fig. 3. Typing rules for expressions (excerpt)

$$\begin{array}{c}
\frac{\Gamma \vdash \mathbf{e}_1 : \text{Tz} \quad \Gamma \vdash \mathbf{e}_2 : \text{Puk} \quad \Gamma \vdash \mathbf{e}_3 : \text{Puk} \quad \Gamma \vdash \mathbf{e}_4 : \text{Unit} \quad \Gamma \vdash \mathbf{e}_5 : \text{Tz}}{\Gamma \vdash \text{transfer } \mathbf{e}_1 \text{ from } \mathbf{e}_2 \text{ to } \mathbf{e}_3 \text{ arg } \mathbf{e}_4 \text{ fee } \mathbf{e}_5 : \text{Oph } \top \ \top} \\
\\
\frac{\Gamma \vdash \mathbf{e}_1 : \text{Tz} \quad \Gamma \vdash \mathbf{e}_2 : \text{Puk} \quad \Gamma \vdash \mathbf{e}_3 : \text{Cont } T_p \ T_s \quad \Gamma \vdash \mathbf{e}_4 : T_p \quad \Gamma \vdash \mathbf{e}_5 : \text{Tz}}{\Gamma \vdash \text{transfer } \mathbf{e}_1 \text{ from } \mathbf{e}_2 \text{ to } \mathbf{e}_3 \text{ arg } \mathbf{e}_4 \text{ fee } \mathbf{e}_5 : \text{Oph } \top \ \top} \\
\\
\frac{\Gamma \vdash \mathbf{e}_1 : \text{Tz} \quad \Gamma \vdash \mathbf{e}_2 : \text{Puk} \quad \Gamma \vdash \mathbf{e}_3 : \text{Code } T_p \ T_s \quad \Gamma \vdash \mathbf{e}_4 : T_s \quad \Gamma \vdash \mathbf{e}_5 : \text{Tz}}{\Gamma \vdash \text{originate contract transferring } \mathbf{e}_1 \text{ from } \mathbf{e}_2 \text{ running } \mathbf{e}_3 \text{ init } \mathbf{e}_4 \text{ fee } \mathbf{e}_5 : \text{Oph } T_p \ T_s} \\
\\
\frac{\Gamma \vdash \mathbf{e} : \text{Addr}}{\Gamma \vdash \text{balance } \mathbf{e} : \text{Tz}} \quad \frac{\Gamma \vdash \mathbf{e} : \text{Oph } T \ U}{\Gamma \vdash \text{status } \mathbf{e} : \text{Status}} \quad \frac{\Gamma \vdash \mathbf{e} : \text{Cont } T_p \ T_s}{\Gamma \vdash \text{storage } \mathbf{e} : T_s} \\
\\
\frac{\Gamma \vdash \mathbf{e} : \text{Oph } T \ U \quad T \neq \top \quad U \neq \top}{\Gamma \vdash \text{contract } \mathbf{e} : \text{Cont } T \ U}
\end{array}$$

Fig. 4. Typing rules for blockchain operations and queries

When an operation is started on a node, it enters a *pool* as a pending operation. A pending operation is either dismissed after some time or promoted to an included operation, which has become a permanent part of the blockchain.

The pool is a partial map $\mathbf{P} = \text{Oph} \hookrightarrow \text{Op} \times \text{Time} \times \text{Status}$ where

$$\text{Status} = \text{pending} + \text{included} + \text{Time} + \text{timeout}$$

such that if $\mathbf{P}(\text{oph}) = \langle \text{op}, \mathbf{t}, \text{st} \rangle$ is defined, then **oph** is the public hash of the operation, **op** is the operation, **t** is the time when the operation was injected, and **st** is either pending, included **t'**, or timeout. A pool \mathbf{P} is *well-formed* if, for all **oph**, $\mathbf{P}(\text{oph}) = \langle \text{op}, \mathbf{t}, \text{included } \mathbf{t}' \rangle$ implies $\mathbf{t}' \geq \mathbf{t}$ and **oph** = genOpHash(**op**, **t**).

A *pending operation* is represented by **oph** $\mapsto \langle \text{op}, \mathbf{t}, \text{pending} \rangle$. Once the operation is accepted, it changes its status to included: **oph** $\mapsto \langle \text{op}, \mathbf{t}, \text{included } \mathbf{t}' \rangle$, where $\mathbf{t}' \geq \mathbf{t}$ is when the operation was included in the blockchain. The operation may also be dropped at any time, which is represented by **oph** $\mapsto \langle \text{op}, \mathbf{t}, \text{timeout} \rangle$. There are several causes for dropping, primarily timeout or overflow of the pending pool which is limited in size in the implementation.

In summary, the *state of a blockchain* is a tuple $\mathbf{B} = [\mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}]$ where \mathbf{P} is a pool of operations, \mathbf{M} is a map of managers, \mathbf{C} is a map of contractors, and **t** is the current time.

We often use the dot notation to project a component from a tuple. For instance, we write $\mathbf{B}.\mathbf{M}$ to access the managers component.

A *blockchain configuration* has the form $\mathbf{B}[\mathbf{N}_1, \dots, \mathbf{N}_n]$, for some $n > 0$, where \mathbf{B} is a blockchain and the \mathbf{N}_i are local nodes, for $1 \leq i \leq n$. In a *well-formed configuration*, the accounts on the local nodes are all different and each local account has a manager in \mathbf{B} :

1. for all $1 \leq i < j \leq n$, $\mathbf{N}_i.\mathbf{A} \cap \mathbf{N}_j.\mathbf{A} = \emptyset$;
2. for all $1 \leq i \leq n$, $\forall a \in \mathbf{N}_i.\mathbf{A} \implies a.\text{puk} \in \text{dom}(\mathbf{B}.\mathbf{M})$.

4 Operational Semantics

The operational semantics is defined by several kinds of transitions:

1. \longrightarrow_E single-step evaluation of an expression in a local node,
2. \longrightarrow_N internal transitions of a node,
3. \longrightarrow_B transitions of the blockchain state,
4. \longrightarrow blockchain system transitions.

Evaluation of expressions is standard for call-by-value lambda calculus defined using evaluation contexts $\underline{\mathbf{E}}[]$. Figure 5 shows some of the reduction rules. The internal transitions of a node are just evaluation of expressions.

$$\frac{\text{NODE-EVAL} \quad \mathbf{e} \longrightarrow_E \mathbf{e}'}{[\underline{\mathbf{E}}[\mathbf{e}] :: \bar{\mathbf{e}}, \mathbf{A}] \longrightarrow_N [\underline{\mathbf{E}}[\mathbf{e}'] :: \bar{\mathbf{e}}, \mathbf{A}]}$$

$$\begin{array}{c}
\frac{}{\underline{E}[(\lambda x.e)v] \longrightarrow_E \underline{E}[e[v/x]]} \quad \frac{}{\underline{E}[\text{try } v \text{ except } e] \longrightarrow_E \underline{E}[v]} \\
\frac{T <: U}{\underline{E}[(v : T \Rightarrow U)] \longrightarrow_E \underline{E}[v]} \quad \frac{\text{try } \notin \underline{F}[]}{\underline{E}[\text{try } \underline{F}[\text{raise } v] \text{ except } e] \longrightarrow_E \underline{E}[e \ v]}
\end{array}$$

Fig. 5. Select expression reduction rules (pattern matching omitted)

$$\begin{array}{c}
\text{NODE-INJECT} \\
\frac{\langle \mathbf{pak}, \mathbf{puk} \rangle \in \mathbf{A} \quad \text{chkBal}(\mathbf{M}, \mathbf{puk}, \mathbf{nt}, \mathbf{fee}) \quad \text{chkArg}(\mathbf{C}, \mathbf{puh}, \mathbf{p}) \quad \text{chkCount}(\mathbf{M}, \mathbf{puk}) \quad \text{chkPuh}(\mathbf{C}, \mathbf{puh}) \quad \text{chkFee}(\mathbf{C}, \mathbf{puh}, \mathbf{p}, \mathbf{fee})}{\mathbf{oph} = \text{genOpHash}(\mathbf{op}, \mathbf{t}) \quad \mathbf{op} = \text{transfer } \mathbf{nt} \text{ from } \mathbf{puk} \text{ to } \mathbf{puh} \text{ arg } \mathbf{p} \text{ fee } \mathbf{fee}} \\
\frac{}{[\underline{E}[\mathbf{op}] :: \bar{e}, \mathbf{A}] \parallel [\mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow [\underline{E}[\mathbf{oph}] :: \bar{e}, \mathbf{A}] \parallel [\mathbf{oph} \mapsto \langle \mathbf{op}, \mathbf{t}, \text{pending} \rangle :: \mathbf{P}, \text{updCount}(\mathbf{M}, \mathbf{puk}, \text{True}), \mathbf{C}, \mathbf{t}]} \\
\\
\text{NODE-REJECT} \\
\frac{\neg \text{chkBal}(\mathbf{B}, \mathbf{M}, \mathbf{op.puk}, \mathbf{op.nt}, \mathbf{op.fee})}{[\underline{E}[\mathbf{op}] :: \bar{e}, \mathbf{A}] \parallel \mathbf{B} \longrightarrow [\underline{E}[\text{raise xBal}] :: \bar{e}, \mathbf{A}] \parallel \mathbf{B}} \\
\\
\text{BLOCK-ACCEPT} \\
\frac{\mathbf{op} = \text{transfer } \mathbf{nt} \text{ from } \mathbf{puk} \text{ to } \mathbf{puh} \text{ arg } \mathbf{p} \text{ fee } \mathbf{fee} \quad \mathbf{t} - \hat{\mathbf{t}} \leq 60}{[\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{pending} \rangle :: \mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow_B [\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{included } \mathbf{t} \rangle :: \mathbf{P}, \text{updSucc}(\mathbf{M}, \mathbf{puk}, \mathbf{nt}, \mathbf{fee}), \text{updConstr}(\mathbf{C}, \mathbf{puh}, \mathbf{nt}, \mathbf{p}), \mathbf{t} + 1]} \\
\\
\text{BLOCK-TIMEOUT} \\
\frac{\mathbf{t} - \hat{\mathbf{t}} > 60}{[\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{pending} \rangle :: \mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow_B [\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{timeout} \rangle :: \mathbf{P}, \text{updCount}(\mathbf{M}, \mathbf{op.puk}, \text{False}), \mathbf{C}, \mathbf{t}]}
\end{array}$$

Fig. 6. Lifecycle transitions of a transaction

The remaining transitions affect a local node in the context of the blockchain. To this end, any local node may be selected.

$$\begin{array}{ccc}
\text{CONFIG-SYSTEM} & \text{CONFIG-NODE} & \text{CONFIG-BLOCK} \\
\frac{\mathbf{N} \parallel \mathbf{B} \longrightarrow \mathbf{N}' \parallel \mathbf{B}'}{\mathbf{B}[\mathbf{N} :: \bar{\mathbf{N}}] \longrightarrow \mathbf{B}'[\mathbf{N}' :: \bar{\mathbf{N}}]} & \frac{\mathbf{N} \longrightarrow_N \mathbf{N}'}{\mathbf{B}[\mathbf{N} :: \bar{\mathbf{N}}] \longrightarrow \mathbf{B}[\mathbf{N}' :: \bar{\mathbf{N}}]} & \frac{\mathbf{B} \longrightarrow_B \mathbf{B}'}{\mathbf{B}[\bar{\mathbf{N}}] \longrightarrow \mathbf{B}'[\bar{\mathbf{N}}]}
\end{array}$$

Figure 6 shows the transitions to start and finalize a contract invocation. NODE-INJECT affects a local node and the blockchain. It nondeterministically selects a program that wants to do a transfer operation. It checks whether the sender of the transfer is a valid local account, whether the balance is sufficient to pay the fee and the transferred amount, whether there is an active transition for this sender (chkCount), whether the public hash is associated with a smart contract on the blockchain, whether the type of the input parameter matches with the smart contract's parameter type (chkArg), and whether the fee is suf-

$$\begin{array}{c}
\text{CONTRACT-YES} \\
\frac{\vdash_C \text{code} : \text{Pair } T \ U \quad \mathbf{B.C}(\mathbf{puh}) = \langle \text{code}, \tilde{t}, \mathbf{nt}', s' \rangle}{[\underline{\mathbf{E}}[(\mathbf{puh} : \text{Puh} \Rightarrow \text{Cont } T)] :: \bar{\mathbf{e}}, \mathbf{A}]\|\mathbf{B} \longrightarrow [\underline{\mathbf{E}}[\mathbf{puh}] :: \bar{\mathbf{e}}, \mathbf{A}]\|\mathbf{B}} \\
\\
\text{CONTRACT-NO} \\
\frac{\mathbf{B.C}(\mathbf{puh}) = \langle \text{code}, \tilde{t}, \mathbf{nt}', s' \rangle \Rightarrow \vdash_C \text{code} : \text{Pair } T' \ U \wedge T \neq T'}{[\underline{\mathbf{E}}[(\mathbf{puh} : \text{Puh} \Rightarrow \text{Cont } T)] :: \bar{\mathbf{e}}, \mathbf{A}]\|\mathbf{B} \longrightarrow [\underline{\mathbf{E}}[\text{raise xPrg}] :: \bar{\mathbf{e}}, \mathbf{A}]\|\mathbf{B}}
\end{array}$$

Fig. 7. Cast reductions (excerpt)

ficient. If these conditions are fulfilled, the transition forges an operation hash and returns it to the local node. The pending operation enters the pool and the sender's counter is set to indicate an ongoing transition.

We give just one example **NODE-REJECT** of the numerous transitions that cover the cases where one of the premises of **NODE-INJECT** is not fulfilled. Each of them raises an exception that describes which condition was violated.

Acceptance or rejection of a pending operation happens on the blockchain independent of any local node. In our model, these transitions are nondeterministic so that acceptance can happen any time in the next 60 cycles **BLOCK-ACCEPT**. Afterwards, a pending operation can only time out **BLOCK-TIMEOUT**. If the transaction is accepted, then the sender's counter is reset, the balances of sender is adjusted (**updSucc**), the smart contract's storage and balance are updated (**updConstr**), and the time stamp increases.

Whereas **NODE-INJECT** and **BLOCK-ACCEPT** are particular to the transfer operation, the timeout transition applies to all operations. It just changes the state of the operation and resets the sender's counter, thus rolling back the transaction.

4.1 Cast Reductions

Figure 7 contains the most interesting example of cast reductions, from a public hash to a typed contract. These reductions force the local node to obtain information from the blockchain. The cast succeeds on **puh** ('**CONTRACT-YES**'), if there is a contractor for **puh** such that the stored code has the parameter type expected by the cast. The cast fails ('**CONTRACT-NO**'), if **puh** is invalid or if the types do not match.

4.2 Smart Contracts

The invocation of smart contracts is similar to a transfer, so we elide the details. Figure 8 contains the transition **BLOCK-ORIGINATE** to originate a smart contract. The basic scheme is similar to the transfer. The preconditions for the operation are checked, but there are extra preconditions for origination: the program must be well-formed and typed, the initial storage value must match its type. The operation ends up in the pool in pending status.

BLOCK-ORIGINATE

$$\frac{\begin{array}{l} \langle \mathbf{pak}, \mathbf{puk} \rangle \in \mathbf{A} \\ \text{chkBal}(\mathbf{M}, \mathbf{puk}, \mathbf{nt}, \mathbf{fee}) \quad \text{chkCount}(\mathbf{M}, \mathbf{puk}) \quad \text{chkPrg}(\mathbf{code}) \\ \text{chkFee}(\mathbf{code}, \mathbf{s}, \mathbf{nt}, \mathbf{fee}) \quad \text{chkInit}(\mathbf{code}, \mathbf{s}) \quad \mathbf{oph} = \text{genOpHash}(\mathbf{op}, \mathbf{t}) \\ \mathbf{op} = \text{originate contract transferring } \mathbf{nt} \text{ from } \mathbf{puk} \text{ running } \mathbf{code} \text{ init } \mathbf{s} \text{ fee } \mathbf{fee} \end{array}}{[\underline{\mathbf{E}}[\mathbf{op}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow [\underline{\mathbf{E}}[\mathbf{oph}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{oph} \mapsto \langle \mathbf{op}, \mathbf{t}, \text{pending} \rangle :: \mathbf{P}, \text{updCount}(\mathbf{M}, \mathbf{puk}, \text{True}), \mathbf{C}, \mathbf{t}]}$$

BLOCK-ORIGINATE-ACCEPT

$$\frac{\begin{array}{l} \mathbf{op} = \text{originate contract transferring } \mathbf{nt} \text{ from } \mathbf{puk} \text{ running } \mathbf{code} \text{ init } \mathbf{s} \text{ fee } \mathbf{fee} \\ \mathbf{puh} = \text{genHash}(\mathbf{code}, \mathbf{t}) \quad \mathbf{t} - \hat{\mathbf{t}} \leq 60 \end{array}}{[\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{pending} \rangle :: \mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow_B [\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{included } \hat{\mathbf{t}} \rangle :: \mathbf{P}, \text{updSucc}(\mathbf{M}, \mathbf{puk}, \mathbf{nt}, \mathbf{fee}), \mathbf{puh} \mapsto \langle \mathbf{code}, \mathbf{t}, \mathbf{nt}, \mathbf{s} \rangle :: \mathbf{C}, \mathbf{t} + 1]}$$

BLOCK-ACCEPT-QUERY

$$\frac{\begin{array}{l} \mathbf{op} = \text{originate contract transferring } \mathbf{nt} \text{ from } \mathbf{puk} \text{ running } \mathbf{code} \text{ init } \mathbf{s} \text{ fee } \mathbf{fee} \\ \mathbf{P}(\mathbf{oph}) = \langle \mathbf{op}, \hat{\mathbf{t}}, \text{included } \hat{\mathbf{t}} \rangle \quad \mathbf{puh} = \text{genHash}(\mathbf{code}, \hat{\mathbf{t}}) \end{array}}{[\underline{\mathbf{E}}[\text{contract } \mathbf{oph}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow [\underline{\mathbf{E}}[\mathbf{puh}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}]}$$

Fig. 8. Smart contract origination

QUERY-BALANCE-IMPLICIT

$$\frac{\mathbf{B.M}(\mathbf{puk}) = \langle \mathbf{bal}, \mathbf{cnt} \rangle}{[\underline{\mathbf{E}}[\text{balance } \mathbf{puk}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B} \longrightarrow [\underline{\mathbf{E}}[\mathbf{bal}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B}}$$

QUERY-BALANCE-FAIL

$$\frac{\mathbf{puk} \notin \text{dom}(\mathbf{B.M})}{[\underline{\mathbf{E}}[\text{balance } \mathbf{puk}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B} \longrightarrow [\underline{\mathbf{E}}[\text{raise xPub}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B}}$$

Fig. 9. Example queries

Acceptance of origination is slightly different as for transfers as shown in BLOCK-ACCEPT. We calculate the public hash **puh** of the contract from the code and the current time stamp and create a new contractor at that address.

We obtain the handle of the contract through a query, once the contract is accepted on the blockchain in BLOCK-ACCEPT-QUERY. The query's argument is the operation hash, which is used to obtain the code and the time stamp of its acceptance. From this information, we can re-calculate the public hash.

4.3 Queries

We conclude with two example transitions for a simple query in Figure 9. To obtain the balance of an implicit account **puk**, we obtain the account info from the manager and extract the balance (QUERY-BALANCE-IMPLICIT). If the account

is unknown, then we raise an exception (QUERY-BALANCE-FAIL). Other queries are implemented analogously.

5 Properties

Having defined our execution model, we proceed to prove properties of the combined systems that ensure type-safe interaction between programs and the blockchain.

5.1 Properties of blockchain state transitions

One interesting property we wish to prove is that the execution of a program that starts with valid references to accounts, operations, and contracts is not corrupted by a transition.

Proposition 1. *The following properties are preserved by a step on a well-formed configuration $[\bar{e}, \mathbf{A}] \parallel \mathbf{B}$:*

- for all \mathbf{oph} in \bar{e} , $\mathbf{oph} \in \text{dom}(\mathbf{B.P})$,
- for all \mathbf{puk} in \bar{e} , $\mathbf{puk} \in \text{dom}(\mathbf{B.M})$,
- for all \mathbf{puh} in \bar{e} , $\mathbf{puh} \in \text{dom}(\mathbf{B.C})$.

Proposition 2. *If $[P, M, C, t] \longrightarrow_B [P', M', C', t']$, then*

1. $t \leq t'$
2. $\text{dom}(\mathbf{P}) \subseteq \text{dom}(\mathbf{P}')$
3. *invariant for the pool: if $\mathbf{P}(\mathbf{oph}) = \langle \mathbf{op}, \hat{t}, \mathbf{st} \rangle$, then $\mathbf{oph} = \text{genOpHash}(\mathbf{op}, \hat{t})$.*
4. *for all $\mathbf{oph} \in \text{dom}(\mathbf{P})$, if $\mathbf{P}(\mathbf{oph}) = \langle \mathbf{op}, \hat{t}, \mathbf{st} \rangle$, then either*
 - $\mathbf{P}'(\mathbf{oph}) = \mathbf{P}(\mathbf{oph})$; or
 - $\mathbf{st} = \text{pending}$ and $\mathbf{P}'(\mathbf{oph}) = \langle \mathbf{op}, \hat{t}, \text{timeout} \rangle$; or
 - $\mathbf{st} = \text{pending}$, $t - \hat{t} \leq 60$, $\mathbf{P}'(\mathbf{oph}) = \langle \mathbf{op}, \hat{t}, \text{included } t \rangle$, and $t' = t + 1$.
5. *for all $\mathbf{oph} \in \text{dom}(\mathbf{P})$ and $\mathbf{P}(\mathbf{oph}) = \langle \mathbf{op}, \hat{t}, \mathbf{st} \rangle$,*
 - *if $\mathbf{st} = \text{pending}$ and $\mathbf{M}(\mathbf{op.puk}) = \langle \mathbf{bal}, \mathbf{cnt} \rangle$ then $\mathbf{cnt.b} = \text{True}$ and $\mathbf{bal} \geq \mathbf{op.nt} + \mathbf{op.fee}$;*
 - *if $\mathbf{st} = \text{included } \hat{t}$, then $\hat{t} < t'$.*
6. $\text{dom}(\mathbf{M}) \subseteq \text{dom}(\mathbf{M}')$
7. *for all $\mathbf{puk} \in \text{dom}(\mathbf{M})$*
 - if $\mathbf{M}(\mathbf{puk}) = \langle \mathbf{bal}, \mathbf{cnt} \rangle$, then $\mathbf{M}'(\mathbf{puk}) = \langle \mathbf{bal}', \mathbf{cnt}' \rangle$ and*
 - *if $\mathbf{cnt.b} = \text{True}$ and $\mathbf{cnt'.b} = \text{False}$, then $\mathbf{cnt'.n} \in \{\mathbf{cnt.n}, \mathbf{cnt.n} + 1\}$,*
 - *otherwise $\mathbf{cnt.n} = \mathbf{cnt'.n}$*
 - *If $\mathbf{cnt.n} = \mathbf{cnt'.n}$, then $\mathbf{bal} = \mathbf{bal}'$.*
8. $\text{dom}(\mathbf{C}) \subseteq \text{dom}(\mathbf{C}')$
 - *for all $\mathbf{puh} \in \text{dom}(\mathbf{C})$, $\mathbf{C}(\mathbf{puh}).\text{code} = \mathbf{C}'(\mathbf{puh}).\text{code}$*
9. *invariant for contractors: for all $\mathbf{puh} \in \text{dom}(\mathbf{C})$,*
 - $\mathbf{C}(\mathbf{puh}) = \langle \mathbf{code}, \hat{t}, \mathbf{bal}, \mathbf{storage} \rangle$ *implies that $\mathbf{puh} = \text{genHash}(\mathbf{code}, \hat{t})$.*

Establishing items 4 and 7 relies on the preimage resistance of the various hash functions used to calculate operation hashes and public hashes: we always feed a fresh timestamp into the hash functions for operations and code. Items 2–5 describe an invariant and the lifecycle of operations. Items 6 and 7 describe the lifecycle of a transfer and items 8 and 9 describe invariants for contractors. The invariants establish the self-verifying property common of blockchain entities.

The proofs of these properties refer to all transitions with the detailed specifications of the related functions, such as `chkCount` and `updSucc`. Due to page limitations, not all transitions and their associated functions are presented in this paper, so the full proofs will be provided in an upcoming technical report. In this paper, we only provide the proofs for Proposition 2 at items 4 and 7.

Proof (4). After feeding into a node, the status of the operation is pending according to the transition `NODE-INJECT`. This operation could either be accepted by the blockchain on the condition that the elapsed time is less than 60 ($\mathbf{t} - \hat{\mathbf{t}} \leq 60$), and then its status is included \mathbf{t} (the transition `BLOCK-ACCEPT`) or it is timed out with the timeout status (`BLOCK-TIMEOUT`). When an operation is accepted or timed out, its status is never changed. Therefore, if $\mathbf{P}(\mathbf{oph}) = \langle \mathbf{op}, \hat{\mathbf{t}}, \mathbf{st} \rangle$, then there are three cases:

- (1) if the operation's status remains the same as \mathbf{st} (still in pending, included or timeout), then we have $\mathbf{P}'(\mathbf{oph}) = \langle \mathbf{op}, \hat{\mathbf{t}}, \mathbf{st} \rangle$. This means $\mathbf{P}'(\mathbf{oph}) = \mathbf{P}(\mathbf{oph})$;
- (2) if the operation's status is pending ($\mathbf{st} = \text{pending}$), and then the operation is timed out, then we have $\mathbf{P}'(\mathbf{oph}) = \langle \mathbf{op}, \hat{\mathbf{t}}, \text{timeout} \rangle$ according to the transition `BLOCK-TIMEOUT`;
- (3) if the operation's status is pending, the time condition is satisfied, and then the operation is accepted, then we have $\mathbf{P}'(\mathbf{oph}) = \langle \mathbf{op}, \hat{\mathbf{t}}, \text{included } \mathbf{t} \rangle$ and $\mathbf{t}' = \mathbf{t} + 1$ because the timestamp is incremented by one according to the transition `BLOCK-ACCEPT`.

From (1), (2) and (3), the item 4 of Proposition 2 is proved.

Proof (7). To prove this point, let us consider the two related functions. The function `updCount`($\mathbf{M}, \mathbf{puk}, \mathbf{b}$) updates the flag of the counter of the account associated with the public key \mathbf{puk} . Its specification is as follows:

$$\text{updCount}(\mathbf{puk} \mapsto \langle \mathbf{bal}, (\mathbf{n}, \hat{\mathbf{b}}) \rangle, \mathbf{b}) = \mathbf{puk} \mapsto \langle \mathbf{bal}, (\mathbf{n}, \mathbf{b}) \rangle$$

The function `updSucc`($\mathbf{M}, \mathbf{puk}, \mathbf{nt}, \mathbf{fee}$) updates the balance and the counter of the account associated with the public key \mathbf{puk} . Its specification is as follows:

$$\text{updSucc}(\mathbf{puk} \mapsto \langle \mathbf{bal}, (\mathbf{n}, \text{True}) \rangle, \mathbf{nt}, \mathbf{fee}) = \mathbf{puk} \mapsto \langle \mathbf{bal} - \mathbf{nt} - \mathbf{fee}, (\mathbf{n} + 1, \text{False}) \rangle$$

if $\mathbf{M}(\mathbf{puk}) = \langle \mathbf{bal}, \mathbf{cnt} \rangle$, then $\mathbf{M}'(\mathbf{puk}) = \langle \mathbf{bal}', \mathbf{cnt}' \rangle$ and we have:

- (1) $\mathbf{cnt}.b = \text{True}$ means that the operation is injected and its status is pending at the time \mathbf{t} according to the transition `NODE-INJECT`. After that, there are only two cases where the counter's flag is reset to `False`. If the operation is accepted, the counter's flag is reset ($\mathbf{cnt}'.b = \text{False}$) according to the transition `BLOCK-ACCEPT` and the counter's value is incremented by 1 according to the specification of the function `updSucc` ($\mathbf{cnt}.n' = \mathbf{cnt}.n + 1$). In another case, if the operation is timed out, the counter's flag is also reset to `False`, but the value of the counter remains the same ($\mathbf{cnt}.n' = \mathbf{cnt}.n$) according to the transition `BLOCK-TIMEOUT`. That is, if $\mathbf{cnt}.b = \text{True}$ and $\mathbf{cnt}'.b = \text{False}$, then $\mathbf{cnt}.n' \in \{\mathbf{cnt}.n, \mathbf{cnt}.n + 1\}$;
- (2) otherwise, if the operation is still pending, the counter's value remains the same. This means $\mathbf{cnt}.n = \mathbf{cnt}'.n$;
- (3) and then $\mathbf{cnt}.n = \mathbf{cnt}'.n$ means that the operation is either still pending or it has timed out. Therefore, the balance of the account remains the same because the balance is only changed when the operation is accepted. This means $\mathbf{bal} = \mathbf{bal}'$.

From (1), (2) and (3), the item 7 of Proposition 2 is proved.

5.2 Typing related properties

To describe the typing of contracts we maintain an environment $\Delta ::= \cdot \mid \mathbf{puh} : T, \Delta$ that associates a public hash with a type. We define typing for blockchains, local nodes, and configurations.

$$\frac{\begin{array}{c} \text{dom}(\Delta) = \text{dom}(\mathbf{B.C}) \quad (\forall \mathbf{puh} \in \text{dom}(\Delta)) \quad \Delta(\mathbf{puh}) = \text{Pair } T_p \ T_s \\ \vdash_C \mathbf{B.C}(\mathbf{puh}).\text{code} : \text{Pair } T_p \ T_s \quad \vdash_V \mathbf{B.C}(\mathbf{puh}).\text{storage} : T_s \end{array}}{\Delta \vdash \mathbf{B}}$$

The type for a hash is a pair type, which coincides with the type of the code stored at that hash. The storage at that hash has the type expected by the code.

$$\frac{\cdot \vdash \mathbf{e}_i : \text{Unit}}{\vdash [\bar{\mathbf{e}}, \mathbf{A}] \text{ ok}} \quad \frac{\Delta \vdash \mathbf{B} \quad \vdash \mathbf{N}_i \text{ ok}}{\Delta \vdash \mathbf{B}[\bar{\mathbf{N}}]}$$

Lemma 1 (Preservation). *If $\mathbf{B}[\bar{\mathbf{N}}] \longrightarrow \mathbf{B}'[\bar{\mathbf{N}}']$ and $\Delta \vdash \mathbf{B}[\bar{\mathbf{N}}]$, then there is some $\Delta' \supseteq \Delta$ such that $\Delta' \vdash \mathbf{B}'[\bar{\mathbf{N}}']$.*

This lemma includes the standard preservation for the lambda calculus part.

Lemma 2 (Progress). *If $\Delta \vdash \mathbf{B}[\bar{\mathbf{N}}]$, then either all expressions in all nodes are unit values or there is a configuration $\mathbf{B}'[\bar{\mathbf{N}}']$ such that $\mathbf{B}[\bar{\mathbf{N}}] \longrightarrow \mathbf{B}'[\bar{\mathbf{N}}']$.*

The consistency lemma says that all committed transactions respect the typing.

Lemma 3 (Consistency). *Consider a blockchain state with $\Delta \vdash [\mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}]$. For all $\mathbf{oph} \in \text{dom}(\mathbf{P})$, if $\mathbf{P}(\mathbf{oph}) = \langle \mathbf{op}, \mathbf{t}, \mathbf{st} \rangle$*

- if $\mathbf{op} = \text{transfer } \mathbf{nt}$ from \mathbf{puk} to \mathbf{puk}' arg $()$ fee \mathbf{fee} , then $\mathbf{puk}, \mathbf{puk}' \in \text{dom}(\mathbf{M})$;
- if $\mathbf{op} = \text{transfer } \mathbf{nt}$ from \mathbf{puk} to \mathbf{puh} arg \mathbf{p} fee \mathbf{fee} , then
 - $\mathbf{puk} \in \text{dom}(\mathbf{M})$ and $\mathbf{puh} \in \text{dom}(\mathbf{C})$,
 - $\vdash_V \mathbf{p} : T_p$ where $\Delta(\mathbf{puh}) = \text{Pair } T_p \ T_s$;
- if $\mathbf{op} = \text{originate contract transferring } \mathbf{nt}$ from \mathbf{puk} running \mathbf{code} init \mathbf{s} fee \mathbf{fee} and $\mathbf{st} = \text{included } \mathbf{t}'$, then
 - $\mathbf{puk} \in \text{dom}(\mathbf{M})$ and $\mathbf{puh} = \text{genHash}(\mathbf{code}, \mathbf{t}') \in \text{dom}(\mathbf{C})$,
 - $\Delta(\mathbf{puh}) = \text{Pair } T_p \ T_s, \vdash_C \mathbf{code} : \text{Pair } T_p \ T_s$ and $\vdash_V \mathbf{s} : T_s$.

Proof. Consider the proof of the second item of Lemma 3, which specifies the property on type for a smart contract invocation. A smart contract call \mathbf{op} has the form $\text{transfer } \mathbf{nt}$ from \mathbf{puk} to \mathbf{puh} arg \mathbf{p} fee \mathbf{fee} . If $\mathbf{P}(\mathbf{oph}) = \langle \mathbf{op}, \hat{\mathbf{t}}, \mathbf{st} \rangle$, then the operation \mathbf{op} is injected into the node. According to the transition NODE-INJECT for a smart contract invocation, the public key is valid and the public hash must be associated with a smart contract on the blockchain. This means $\mathbf{puk} \in \text{dom}(\mathbf{M})$ and $\mathbf{puh} \in \text{dom}(\mathbf{C})$. Moreover, the chkArg function checks whether the type of the input parameter \mathbf{p} matches the parameter type of the smart contract. If the casted type of the smart contract is $\Delta(\mathbf{puh}) = \text{Pair } T_p \ T_s$, then the type of the parameter must be T_p . This means $\vdash_V \mathbf{p} : T_p$. Therefore, this item is proved.

6 Related Work

The inability to access external data sources limits the potential of smart contracts. Oracles [13,6,3] can help overcome this limitation by providing a bridge between the outside sources and the blockchain network. A blockchain oracle is used to provide external data for smart contracts. When the external data is available, an oracle invokes a smart contract with that information. The invocation can conveniently be made through a programmatic interface. There has been extensive research on providing oracle solutions for blockchain. Adler et al [11] propose a framework to explain blockchain oracles and various key aspects of oracles. This framework aims to provide developers with a guide for incorporating oracles into blockchain-based applications. The main problems with using a blockchain oracle are the untrusted data provided maliciously or inaccurately [1]. Ma et al [10] propose an oracle equipped with verification and disputation mechanisms. Similarly, Lo et al [9] provide a framework for performing reliability analysis of various blockchain oracle platforms.

Current blockchains such as Ethereum [5] and Tezos [8] often offer RPC APIs and use loosely structured data, such as a JSON-based format that is difficult for a programmatic program to handle. As a result, there is increasing work to provide better programmatic interfaces to blockchains. Web3.js [15] provides an Ethereum JavaScript API and offers Java Script users a convenient interface to interact with the Ethereum blockchain. Later, Web3.py [16], derived from Web3.js, is developed to provide a Python library for interacting with Ethereum. Our typed API not only supports for programmatic programs, but also provides verifiable interaction with the Tezos smart contract platform.

7 Conclusion

We present a first step towards a typed API for smart contracts on the Tezos blockchain. Our formalization enables us to establish basic properties of the interaction between ordinary programs and smart contracts. We see ample scope for future work to provide a higher-level interface that exploits the similarities between blockchain programming and concurrent programs. The next step will be to formalize the typing-related results. The formalization could connect with the Mi-Cho-Coq formalization of Michelson contracts [4]. In the end, we would like to state and prove properties of a system that contains OCaml code (multi-threaded or distributed) connected to Michelson contracts on the Tezos blockchain via the typed API.

References

1. Al-Breiki, H., Rehman, M.H.U., Salah, K., Svetinovic, D.: Trustworthy blockchain oracles: Review, comparison, and open research challenges. *IEEE Access* **8**, 85675–85685 (2020)
2. Allombert, V., Bourgoïn, M., Tesson, J.: Introduction to the Tezos blockchain. In: 2019 International Conference on High Performance Computing Simulation (HPCS). pp. 1–10 (2019). <https://doi.org/10.1109/HPCS48598.2019.9188227>
3. Beniiche, A.: A study of blockchain oracles (2020)
4. Bernardo, B., Cauderlier, R., Hu, Z., Pesin, B., Tesson, J.: Mi-Cho-Coq, a framework for certifying Tezos smart contracts. In: Formal Methods. FM 2019 International Workshops, Revised Selected Papers, Part I. Lecture Notes in Computer Science, vol. 12232, pp. 368–379. Springer (2019). https://doi.org/10.1007/978-3-030-54994-7_28
5. Buterin, V.: A next-generation smart contract and decentralized application platform (2013), <https://ethereum.org/en/whitepaper/>
6. Caldarelli, G.: Understanding the blockchain oracle problem: A call for action. *Information* **11**(11) (2020)
7. Ethereum JSON-RPC API (2021), <https://ethereum.org/en/developers/docs/apis/json-rpc/>
8. Goodman, L.: Tezos-a self-amending crypto-ledger (2014), <https://www.tezos.com/static/papers/white-paper.pdf>
9. Lo, S.K., Xu, X., Staples, M., Yao, L.: Reliability analysis for blockchain oracles. *Computers & Electrical Engineering* **83**, 106582 (2020)
10. Ma, L., Kaneko, K., Sharma, S., Sakurai, K.: Reliable decentralized oracle with mechanisms for verification and disputation. In: 2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW). pp. 346–352 (2019)
11. Mammadzada, K., Iqbal, M., Milani, F., García-Bañuelos, L., Matulevičius, R.: Blockchain Oracles: A Framework for Blockchain-Based Applications, pp. 19–34. Springer Verlag (09 2020)
12. Michelson: The language of smart contracts in Tezos, <https://tezos.gitlab.io/alpha/michelson.html>
13. Mühlberger, R., Bachhofner, S., Castelló Ferrer, E., Di Ciccio, C., Weber, I., Wöhrer, M., Zdun, U.: Foundational oracle patterns: Connecting blockchain to

- the off-chain world. Business Process Management: Blockchain and Robotic Process Automation Forum p. 35–51 (2020)
14. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), <https://www.tezos.com/static/papers/white-paper.pdf>
 15. Vogelsteller, F., Kotewicz, M., Wilcke, J., Oance, M.: web3.js - Ethereum JavaScript API, <https://web3js.readthedocs.io/en/v1.3.4/>
 16. Vogelsteller, F., Kotewicz, M., Wilcke, J., Oance, M.: web3.py - a Python library for interacting with Ethereum, <https://web3py.readthedocs.io/en/stable/>

A Type Soundness

Lemma 4 (Canonical forms). *Given a set of local accounts \mathbf{A} , a blockchain \mathbf{B} , and a typed value $\cdot \vdash v : T$.*

- If $T = \text{Puh}$, then $v = \mathbf{puh}$ and $\mathbf{puh} \in \text{dom}(\mathbf{B.C})$.
- If $T = \text{Puk}$, then $v = \mathbf{puk}$ and $\exists \mathbf{pak}$ such that $(\mathbf{pak}, \mathbf{puk}) \in \mathbf{A}$.
- If $T = \text{Addr}$, then v is \mathbf{puh} or \mathbf{puk} .
- If $T = \text{Cont } T_p \ T_s$, then $v = \mathbf{puh}$ and $\mathbf{puh} \in \text{dom}(\mathbf{B.C})$ and $\mathbf{B.C}(\mathbf{puh}) = (\mathbf{code}, \mathbf{t}, \mathbf{bal}, \mathbf{storage})$ such that $\vdash_C \mathbf{code} : \text{Pair } T_p \ T_s$ and $\vdash_V \mathbf{storage} : T_s$.
- If $T = \text{Code } T \ U$, then $v = \mathbf{code}$ and $\vdash_C \mathbf{code} : \text{Pair } T \ U$.
- If $T = \text{Oph } T \ U$, then $v = \mathbf{oph}$ and $\mathbf{oph} \in \text{dom}(\mathbf{B.P})$ and $\mathbf{B.P}(\mathbf{oph}) = \langle \mathbf{op}, \mathbf{t}, \mathbf{st} \rangle$ where $T = U = \top$ if \mathbf{op} is a transfer and $T = T_p \neq \top, U = T_s \neq \top$ if \mathbf{op} is originate contract transferring \mathbf{nt} from \mathbf{puk} running \mathbf{code} init \mathbf{s} fee \mathbf{fee} and $\vdash_C \mathbf{code} : \text{Pair } T_p \ T_s$.
- If $T = \text{Status}$, then $v \in \{\text{pending}, \text{included}(\mathbf{i}), \text{timeout}\}$.
- If $T = \text{Exc}$, then $v \in \{x\text{Prg}, x\text{Bal}, x\text{Count}, x\text{Fee}, x\text{Pub}, x\text{Puh}, x\text{Arg}, x\text{Init}\}$.
- If $T = \text{Tz}$, then $v = \mathbf{nt}$, a token amount.
- If $T = \top$, then v can be any syntactic value.
- If $T = \text{Int}$, then $v = \mathbf{i}$.
- If $T = \text{Unit}$, then $v = ()$.
- If $T = \text{Bool}$, then $v \in \{\text{True}, \text{False}\}$.
- If $T = \text{Str}$, then $v = \mathbf{s}$, a string.
- If $T = T \rightarrow U$, then $v = \lambda \mathbf{x}. \mathbf{e}$.
- If $T = \text{Pair } T \ U$, then $v = (\mathbf{v}', \mathbf{v}'')$ where $\cdot \vdash \mathbf{v}' : T$ and $\cdot \vdash \mathbf{v}'' : U$ in context \mathbf{A} and \mathbf{B} .
- If $T = \text{List } T$, then either $v = \text{nil}$ or $v = \text{cons } \mathbf{v}' \ \mathbf{v}''$ where $\cdot \vdash \mathbf{v}' : T$ and $\cdot \vdash \mathbf{v}'' : \text{List } T$ in context \mathbf{A} and \mathbf{B} .
- If $T = \text{Or } T \ U$, then either $v = \text{left } \mathbf{v}'$ where $\cdot \vdash \mathbf{v}' : T$ or $v = \text{right } \mathbf{v}''$ where $\cdot \vdash \mathbf{v}'' : U$ in context \mathbf{A} and \mathbf{B} .
- If $T = \text{Option } T$, then $v = \text{none}$ or $v = \text{some } \mathbf{v}'$ where $\cdot \vdash \mathbf{v}' : T$ in context \mathbf{A} and \mathbf{B} .

Lemma 5 (Subterm replacement). *If $\cdot \vdash \underline{\mathbf{E}}[\mathbf{e}] : T$, $\cdot \vdash \mathbf{e} : T'$, and $\cdot \vdash \mathbf{e}' : T'$, then $\cdot \vdash \underline{\mathbf{E}}[\mathbf{e}'] : T$.*

Proof. Induction on evaluation context $\underline{\mathbf{E}}$ making use of the fact that an evaluation context does not bind variables.

Lemma 6 (Preservation for expressions). *If $\cdot \vdash \mathbf{e} : T$ and $\mathbf{e} \rightarrow_E \mathbf{e}'$, then $\cdot \vdash \mathbf{e}' : T$.*

Proof. Standard result: type preservation for simply typed lambda calculus with pairs, sums, and exceptions. Uses Lemma 5 for reductions in evaluation context. See, for instance, Types in Programming Languages by Benjamin Pierce.

Lemma 7 (Preservation). *If $\mathbf{B}[\overline{\mathbf{N}}] \rightarrow \mathbf{B}'[\overline{\mathbf{N}}']$ and $\Delta \vdash \mathbf{B}[\overline{\mathbf{N}}]$, then there is some $\Delta' \supseteq \Delta$ such that $\Delta' \vdash \mathbf{B}'[\overline{\mathbf{N}}']$.*

Proof. The proof is by induction on the reduction relation $\mathbf{B}[\bar{\mathbf{N}}] \longrightarrow \mathbf{B}'[\bar{\mathbf{N}}']$ and inversion of the typing judgments. We only consider the exemplary reductions shown in the paper. We mark all components that belong to the reductum with ' as in \mathbf{N}' .

From $\Delta \vdash \mathbf{B}[\bar{\mathbf{N}}]$ we obtain

$$\Delta \vdash \mathbf{B} \quad (1)$$

$$\vdash \mathbf{N}_i \text{ ok} \quad (2)$$

From (1) we obtain

$$\text{dom}(\Delta) = \text{dom}(\mathbf{B.C}) \quad (3)$$

and $\forall \mathbf{puh} \in \text{dom}(\Delta)$

$$\Delta(\mathbf{puh}) = \text{Pair } T_p \ T_s \quad (4)$$

$$\vdash_C \mathbf{B.C}(\mathbf{puh}).\text{code} : \text{Pair } T_p \ T_s \quad (5)$$

$$\vdash_V \mathbf{B.C}(\mathbf{puh}).\text{storage} : T_s \quad (6)$$

From (2) we obtain, if $\mathbf{N}_i = [\bar{\mathbf{e}}_i, \mathbf{A}_i]$,

$$\cdot \vdash \mathbf{e}_{ij} : \text{Unit} \quad (7)$$

$$\text{Reduction} \frac{\text{CONFIG-NODE} \quad \mathbf{N}_0 \longrightarrow_N \mathbf{N}'_0}{\mathbf{B}[\mathbf{N}_0 :: \bar{\mathbf{N}}] \longrightarrow \mathbf{B}[\mathbf{N}'_0 :: \bar{\mathbf{N}}]} \text{NODE-EVAL}$$

The only possible reduction here is $\frac{\mathbf{e} \longrightarrow_E \mathbf{e}'}{[\mathbf{E}[\mathbf{e}] :: \bar{\mathbf{e}}, \mathbf{A}] \longrightarrow_N [\mathbf{E}[\mathbf{e}'] :: \bar{\mathbf{e}}, \mathbf{A}]}$.

From (7), we know that $\cdot \vdash \mathbf{e} : \text{Unit}$. By Lemma 6, $\cdot \vdash \mathbf{e}' : \text{Unit}$, the types of the $\bar{\mathbf{e}}$ are not affected, hence $\vdash \mathbf{N}'_0 = [\mathbf{E}[\mathbf{e}'] :: \bar{\mathbf{e}}, \mathbf{A}] \text{ ok}$. None of the other nodes changed, neither did \mathbf{B} , so that $\Delta \vdash \mathbf{B}[\mathbf{N}'_0 :: \bar{\mathbf{N}}]$.

$$\text{Reduction} \frac{\text{CONFIG-SYSTEM} \quad \mathbf{N} \parallel \mathbf{B} \longrightarrow \mathbf{N}' \parallel \mathbf{B}'}{\mathbf{B}[\mathbf{N} :: \bar{\mathbf{N}}] \longrightarrow \mathbf{B}'[\mathbf{N}' :: \bar{\mathbf{N}}]}.$$

We need to consider the cases for \longrightarrow .

$$\text{Subcase} \frac{\text{NODE-INJECT} \quad \begin{array}{l} \langle \mathbf{pak}, \mathbf{puk} \rangle \in \mathbf{A} \\ \text{chkBal}(\mathbf{M}, \mathbf{puk}, \mathbf{nt}, \mathbf{fee}) \quad \text{chkArg}(\mathbf{C}, \mathbf{puh}, \mathbf{p}) \quad \text{chkCount}(\mathbf{M}, \mathbf{puk}) \\ \text{chkPuh}(\mathbf{C}, \mathbf{puh}) \quad \text{chkFee}(\mathbf{C}, \mathbf{puh}, \mathbf{p}, \mathbf{fee}) \quad \mathbf{oph} = \text{genOpHash}(\mathbf{op}, \mathbf{t}) \\ \mathbf{op} = \text{transfer } \mathbf{nt} \text{ from } \mathbf{puk} \text{ to } \mathbf{puh} \text{ arg } \mathbf{p} \text{ fee } \mathbf{fee} \end{array}}{[\mathbf{E}[\mathbf{op}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow [\mathbf{E}[\mathbf{oph}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{oph} \mapsto \langle \mathbf{op}, \mathbf{t}, \text{pending} \rangle :: \mathbf{P}, \text{updCount}(\mathbf{M}, \mathbf{puk}, \text{True}), \mathbf{C}, \mathbf{t}]}.$$

Here $\mathbf{N} = [\mathbf{E}[\mathbf{op}] :: \bar{\mathbf{e}}, \mathbf{A}]$. We first check type preservation for the expression part. There are two typing rules for the transfer \mathbf{op} , but only the one for contract invocation applies as the other one requires $\cdot \vdash \mathbf{puh} : \text{Puk}$, which does not hold.

For a contract invocation (specialized to empty environment)

$$\frac{\cdot \vdash \mathbf{nt} : \text{Tz} \quad \cdot \vdash \mathbf{puk} : \text{Puk} \quad \cdot \vdash \mathbf{puh} : \text{Cont } T_p \text{ } T_s \quad \cdot \vdash \mathbf{p} : T_p \quad \cdot \vdash \mathbf{fee} : \text{Tz}}{\cdot \vdash \text{transfer } \mathbf{nt} \text{ from } \mathbf{puk} \text{ to } \mathbf{puh} \text{ arg } \mathbf{p} \text{ fee } \mathbf{fee} : \text{Oph } \top \top} \quad (8)$$

The canonical forms lemma 4 is parameterized over the accounts \mathbf{A} of the local node and the current contractors $\mathbf{B.C}$. Hence, we know that the arguments are legal, which is also checked by the rule.

The reduct returns an operation hash \mathbf{oph} at type $\text{Oph } \top \top$, which places no restrictions on the context of \mathbf{oph} .

Moreover, $\Delta' = \Delta$ and $\mathbf{C}' = \mathbf{C}$ as no new contract is originated.

We conclude with Lemma 5 and reapplying CONFIG-SYSTEM.

$$\text{Subcase} \frac{\text{CONTRACT-YES} \quad \vdash_C \text{code} : \text{Pair } T \text{ } U \quad \mathbf{B.C}(\mathbf{puh}) = \langle \text{code}, \tilde{\mathbf{t}}, \mathbf{nt}', \mathbf{s}' \rangle}{[\mathbf{E}[(\mathbf{puh} : \text{Puh} \Rightarrow \text{Cont } T)] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B} \longrightarrow [\mathbf{E}[\mathbf{puh}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B}}.$$

Immediate using Lemma 4 and Lemma 5.

$$\text{Subcase} \frac{\text{CONTRACT-NO} \quad \mathbf{B.C}(\mathbf{puh}) = \langle \text{code}, \tilde{\mathbf{t}}, \mathbf{nt}', \mathbf{s}' \rangle \Rightarrow \vdash_C \text{code} : \text{Pair } T' \text{ } U \wedge T \neq T'}{[\mathbf{E}[(\mathbf{puh} : \text{Puh} \Rightarrow \text{Cont } T)] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B} \longrightarrow [\mathbf{E}[\text{raise xPrg}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B}}.$$

The typing rule for raise can return any type. Hence, this is immediate by Lemma 5.

$$\text{Subcase} \frac{\text{BLOCK-ORIGINATE} \quad \begin{array}{l} \langle \mathbf{pak}, \mathbf{puk} \rangle \in \mathbf{A} \\ \text{chkBal}(\mathbf{M}, \mathbf{puk}, \mathbf{nt}, \mathbf{fee}) \quad \text{chkCount}(\mathbf{M}, \mathbf{puk}) \quad \text{chkPrg}(\text{code}) \\ \text{chkFee}(\text{code}, \mathbf{s}, \mathbf{nt}, \mathbf{fee}) \quad \text{chkInit}(\text{code}, \mathbf{s}) \quad \mathbf{oph} = \text{genOpHash}(\mathbf{op}, \mathbf{t}) \\ \mathbf{op} = \text{originate contract transferring } \mathbf{nt} \text{ from } \mathbf{puk} \text{ running } \text{code} \text{ init } \mathbf{s} \text{ fee } \mathbf{fee} \end{array}}{[\mathbf{E}[\mathbf{op}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow [\mathbf{E}[\mathbf{oph}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{oph} \mapsto \langle \mathbf{op}, \mathbf{t}, \text{pending} \rangle :: \mathbf{P}, \text{updCount}(\mathbf{M}, \mathbf{puk}, \text{True}), \mathbf{C}, \mathbf{t}]}.$$

Suppose that $\vdash_C \mathbf{code} : \text{Pair } T_p \ T_s$. Then $\cdot \vdash \mathbf{op} : \text{Oph } T_p \ T_s$. But this is the type of the \mathbf{oph} in the reductum as it points to \mathbf{op} in \mathbf{P} . Hence, the result is immediate by Lemma 5.

QUERY-BALANCE-IMPLICIT

$$\text{Subcase } \frac{\mathbf{B.M}(\mathbf{puk}) = \langle \mathbf{bal}, \mathbf{cnt} \rangle}{[\underline{\mathbf{E}}[\text{balance } \mathbf{puk}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B} \longrightarrow [\underline{\mathbf{E}}[\mathbf{bal}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B}}.$$

The reduction replaces $\text{balance } \mathbf{puk}$ of type T_z by \mathbf{bal} of the same type. Hence, the result is immediate by Lemma 5.

QUERY-BALANCE-FAIL

$$\text{Subcase } \frac{\mathbf{puk} \notin \text{dom}(\mathbf{B.M})}{[\underline{\mathbf{E}}[\text{balance } \mathbf{puk}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B} \longrightarrow [\underline{\mathbf{E}}[\text{raise xPub}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel \mathbf{B}}.$$

Immediate by Lemma 5 because raise can have any type.

$$\text{Reduction } \frac{\text{CONFIG-BLOCK} \quad \mathbf{B} \longrightarrow_B \mathbf{B}'}{\mathbf{B}[\mathbf{N}] \longrightarrow \mathbf{B}'[\mathbf{N}]}.$$

We need to consider cases for \longrightarrow_B .

$$\text{Subcase } \frac{\text{BLOCK-ACCEPT} \quad \mathbf{op} = \text{transfer } \mathbf{nt} \text{ from } \mathbf{puk} \text{ to } \mathbf{puh} \text{ arg } \mathbf{p} \text{ fee } \mathbf{fee} \quad \mathbf{t} - \hat{\mathbf{t}} \leq 60}{[\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{pending} \rangle :: \mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow_B [\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{included } \mathbf{t} \rangle :: \mathbf{P}, \text{updSucc}(\mathbf{M}, \mathbf{puk}, \mathbf{nt}, \mathbf{fee}), \text{updConstr}(\mathbf{C}, \mathbf{puh}, \mathbf{nt}, \mathbf{p}), \mathbf{t} + 1]}.$$

No typing-related properties are affected.

$$\text{Subcase } \frac{\text{BLOCK-ORIGINATE-ACCEPT} \quad \mathbf{op} = \text{originate contract transferring } \mathbf{nt} \text{ from } \mathbf{puk} \text{ running } \mathbf{code} \text{ init } \mathbf{s} \text{ fee } \mathbf{fee} \quad \mathbf{puh} = \text{genHash}(\mathbf{code}, \mathbf{t}) \quad \mathbf{t} - \hat{\mathbf{t}} \leq 60}{[\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{pending} \rangle :: \mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow_B [\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{included } \mathbf{t} \rangle :: \mathbf{P}, \text{updSucc}(\mathbf{M}, \mathbf{puk}, \mathbf{nt}, \mathbf{fee}), \mathbf{puh} \mapsto \langle \mathbf{code}, \mathbf{t}, \mathbf{nt}, \mathbf{s} \rangle :: \mathbf{C}, \mathbf{t} + 1]}.$$

This reduction extends \mathbf{C} with a new entry for \mathbf{puh} . To preserve typing, we need to extend Δ with the binding $\mathbf{puh} : \text{Pair } T_p \ T_s$ where $\vdash_C \mathbf{code} : \text{Pair } T_p \ T_s$. The generated code pointer is obtained with a query operation via the operation $\text{hash } \mathbf{oph}$, which is also connected to the parameter and storage types.

$$\text{Subcase } \frac{\text{BLOCK-TIMEOUT} \quad \mathbf{t} - \hat{\mathbf{t}} > 60}{[\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{pending} \rangle :: \mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow_B [\mathbf{oph} \mapsto \langle \mathbf{op}, \hat{\mathbf{t}}, \text{timeout} \rangle :: \mathbf{P}, \text{updCount}(\mathbf{M}, \mathbf{op.puk}, \text{False}), \mathbf{C}, \mathbf{t}]}.$$

No typing-related properties are affected.

Lemma 8 (Progress for expressions). *If $\cdot \vdash e : T$, then either*

- *e is a value,*
- *$e \longrightarrow_E e'$, or*
- *$e = \underline{E}[e']$ is a blockchain operation in an evaluation context:*
 - *$e' = \text{transfer } \mathbf{nt} \text{ from } \mathbf{puk} \text{ to } \mathbf{puh} \text{ arg } \mathbf{p} \text{ fee } \mathbf{fee};$*
 - *$e' = \text{originate contract transferring } \mathbf{nt} \text{ from } \mathbf{puk} \text{ running code init } \mathbf{s} \text{ fee } \mathbf{fee};$*
 - *$e' = \mathbf{qop } \mathbf{v};$*
 - *$e' = (\mathbf{v} : T \Rightarrow U)$ where $U <: T$.*

Proof. Standard result: progress for simply type lambda calculus with pairs, sums, and exceptions. Upcasts are resolved by identity reductions. The blockchain operations including downcasts are not handled by the \longrightarrow_E relation.

Lemma 9 (Progress). *If $\Delta \vdash B[\bar{N}]$, then either all expressions in all nodes are unit values or there is a configuration $B'[\bar{N}']$ such that $B[\bar{N}] \longrightarrow B'[\bar{N}']$.*

Proof. From $\Delta \vdash B[\bar{N}]$ we obtain

$$\Delta \vdash \mathbf{B} \quad (9)$$

$$\vdash \mathbf{N}_i \text{ ok} \quad (10)$$

From (9) we obtain

$$\text{dom}(\Delta) = \text{dom}(\mathbf{B.C}) \quad (11)$$

and $\forall \mathbf{puh} \in \text{dom}(\Delta)$

$$\Delta(\mathbf{puh}) = \text{Pair } T_p \ T_s \quad (12)$$

$$\vdash_C \mathbf{B.C}(\mathbf{puh}).\text{code} : \text{Pair } T_p \ T_s \quad (13)$$

$$\vdash_V \mathbf{B.C}(\mathbf{puh}).\text{storage} : T_s \quad (14)$$

From (10) we obtain, if $\mathbf{N}_i = [\bar{\mathbf{e}}_i, \mathbf{A}_i]$,

$$\cdot \vdash \mathbf{e}_{ij} : \text{Unit} \quad (15)$$

For each such \mathbf{e}_{ij} , Lemma 8 yields that either

- \mathbf{e}_{ij} is a value; as it has type Unit, we obtain $\mathbf{e}_{ij} = ()$ by Lemma 4;
- $\mathbf{e}_{ij} \longrightarrow_E \mathbf{e}'_{ij}$, in which case the whole system makes a step; or
- $\mathbf{e}_{ij} = \underline{E}[\mathbf{e}]$ where \mathbf{e} is a blockchain operation.

Subcase $\mathbf{e} = \text{transfer } \mathbf{nt} \text{ from } \mathbf{puk} \text{ to } \mathbf{puh} \text{ arg } \mathbf{p} \text{ fee } \mathbf{fee}$. In this case, the NODE-INJECT reduction is in principle enabled:

NODE-INJECT

$$\frac{\begin{array}{c} \langle \mathbf{pak}, \mathbf{puk} \rangle \in \mathbf{A} \\ \text{chkBal}(\mathbf{M}, \mathbf{puk}, \mathbf{nt}, \mathbf{fee}) \quad \text{chkArg}(\mathbf{C}, \mathbf{puh}, \mathbf{p}) \quad \text{chkCount}(\mathbf{M}, \mathbf{puk}) \\ \text{chkPuh}(\mathbf{C}, \mathbf{puh}) \quad \text{chkFee}(\mathbf{C}, \mathbf{puh}, \mathbf{p}, \mathbf{fee}) \quad \mathbf{oph} = \text{genOpHash}(\mathbf{op}, \mathbf{t}) \\ \mathbf{op} = \text{transfer } \mathbf{nt} \text{ from } \mathbf{puk} \text{ to } \mathbf{puh} \text{ arg } \mathbf{p} \text{ fee } \mathbf{fee} \end{array}}{[\underline{E}[\mathbf{op}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow [\underline{E}[\mathbf{oph}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{oph} \mapsto \langle \mathbf{op}, \mathbf{t}, \text{pending} \rangle :: \mathbf{P}, \text{updCount}(\mathbf{M}, \mathbf{puk}, \text{True}), \mathbf{C}, \mathbf{t}]}$$

Thanks to the canonical forms Lemma 4, we know that $\langle \mathbf{pak}, \mathbf{puk} \rangle \in \mathbf{A}$, $\text{chkArg}(\mathbf{C}, \mathbf{puh}, \mathbf{p})$ holds, and $\text{chkPuh}(\mathbf{C}, \mathbf{puh})$ holds. If one of the remaining checks fails, then one of the **NODE-REJECT** transitions throws an exception, so the configuration steps in every case.

Subcase $\mathbf{e} = \text{originate contract transferring } \mathbf{nt} \text{ from } \mathbf{puk} \text{ running } \mathbf{code} \text{ init } \mathbf{s} \text{ fee } \mathbf{fee}$. In this case, the **BLOCK-ORIGINATE** reduction is in principle enabled:

BLOCK-ORIGINATE

$$\frac{\begin{array}{c} \langle \mathbf{pak}, \mathbf{puk} \rangle \in \mathbf{A} \\ \text{chkBal}(\mathbf{M}, \mathbf{puk}, \mathbf{nt}, \mathbf{fee}) \quad \text{chkCount}(\mathbf{M}, \mathbf{puk}) \quad \text{chkPrg}(\mathbf{code}) \\ \text{chkFee}(\mathbf{code}, \mathbf{s}, \mathbf{nt}, \mathbf{fee}) \quad \text{chkInit}(\mathbf{code}, \mathbf{s}) \quad \mathbf{oph} = \text{genOpHash}(\mathbf{op}, \mathbf{t}) \\ \mathbf{op} = \text{originate contract transferring } \mathbf{nt} \text{ from } \mathbf{puk} \text{ running } \mathbf{code} \text{ init } \mathbf{s} \text{ fee } \mathbf{fee} \end{array}}{\begin{array}{c} [\underline{\mathbf{E}}[\mathbf{op}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{P}, \mathbf{M}, \mathbf{C}, \mathbf{t}] \longrightarrow \\ [\underline{\mathbf{E}}[\mathbf{oph}] :: \bar{\mathbf{e}}, \mathbf{A}] \parallel [\mathbf{oph} \mapsto \langle \mathbf{op}, \mathbf{t}, \text{pending} \rangle :: \mathbf{P}, \text{updCount}(\mathbf{M}, \mathbf{puk}, \text{True}), \mathbf{C}, \mathbf{t}] \end{array}}$$

Thanks to the canonical forms Lemma 4, we know that $\langle \mathbf{pak}, \mathbf{puk} \rangle \in \mathbf{A}$, $\text{chkPrg}(\mathbf{code})$ holds, and $\text{chkInit}(\mathbf{code}, \mathbf{s})$ holds. If one of the remaining checks fails, then one of the **NODE-REJECT** transitions throws an exception, so the configuration steps in every case.

Subcase $\mathbf{e} = \mathbf{qop} \mathbf{v}$. If $\mathbf{e} = \text{balance } \mathbf{v}$, then inversion tells us that $\cdot \vdash \mathbf{v} : \text{Addr}$ and by canonical forms (Lemma 4), it must be that \mathbf{v} has the form \mathbf{puk} or \mathbf{puh} . In any case, the value is a meaningful address for the manager \mathbf{M} . Depending on whether the address is in use, one of the reductions **QUERY-BALANCE-IMPLICIT** or **QUERY-BALANCE-FAIL** can execute. There are further analogous reductions handling the case where $\mathbf{v} = \mathbf{puh}$ and we ask for the balance of a smart contract.

Most queries behave like $\text{balance } \cdot$, except getting a contract handle from an operation hash:

Subcase $\mathbf{e} = \text{contract } \mathbf{v}$. This query is somewhat special as it is handled with reduction **BLOCK-ACCEPT-QUERY**. By inversion and canonical forms (Lemma 4) we know that $\mathbf{v} = \mathbf{oph}$ is a valid operation hash of type $\text{Cont } T \ U$ where $T \neq \top$ and $U \neq \top$.

However, this reduction is conditional on the state of the transaction; it requires the new contract to have status included. If the contract has status timeout, then the query raises an exception, analogous to the **QUERY-BALANCE-FAIL** reduction. If the contract has status pending, then the expression is blocked, but the system can make a step using **BLOCK-ORIGINATE-ACCEPT** that changes the status from pending to included. Alternatively, **BLOCK-TIMEOUT** can make a step to change the status to timeout. In any case, the system as a whole can make a reduction.

Subcase $\mathbf{e} = (\mathbf{v} : T \Rightarrow U)$ where $U <: T$. As an example, we consider the reductions **CONTRACT-YES** and **CONTRACT-NO**, where a cast is applied to a

value of type Puh. By canonical forms, we know that the value has the form $\mathbf{puh} \in \text{dom}(\mathbf{B.C})$. The code pointed to by this hash is checked at run time and results either in a \mathbf{puh} at suitable contract type (-Yes reduction) or in raising an exception (-No reduction).