Imperial College London

BEng Individual Project

Imperial College London

Department of Computing

Chainlog: A Logic-based Smart Contract Language

Author:
Arran O'Sullivan

Supervisor:
Dr. Naranker Dulay

Second Marker:
Prof. Emil Lupu

June 20, 2022

Abstract

Blockchain-based smart contracts are typically expressed in imperative programming languages such as Ethereum's Solidity. The declarative paradigm has been explored to some extent, and some functional programming options have emerged. However, there has yet to be an implementation of a domain-specific logic-based language for writing smart contracts.

Creating such an implementation is the subject of this work. We present Chainlog: a new language and blockchain platform for writing and executing smart contracts in a logic programming style. Contained in this report is a full definition of the Chainlog language's syntax and semantics; a formal specification of our blockchain, the Chainlog Platform; and a detailed description of our implementation of both.

A qualitative evaluation of the Chainlog language finds it to be expressive and close to high-level specification, resulting in improved comprehensibility and correctness compared to Solidity. An analysis of the Chainlog Platform's security demonstrates the blockchain to be robust to failure and attacks, while maintaining transparency, immutability and decentralisation.

Acknowledgements

My sincere gratitude goes to my supervisor, Dr. Naranker Dulay, for his guidance, insight and enthusiasm in this endeavour.

I also give thanks to Prof. Emil Lupu and Dr. Krysia Broda for the valuable feedback they have provided at various points during the project.

Finally, my acknowledgements would not be complete without recognising my family, to whom I am eternally grateful for their unwavering support throughout my degree.

Contents

1 Introduction **5**

1.1 Contributions 6

2 Background **7**

2.1 Blockchain 7

2.1.1 Replication and Decentralisation 8

2.1.2 Blocks 8

2.1.3 Transactions and Queries 9

2.1.4 Consensus 9

2.2 Smart Contracts 10

2.2.1 Determinism 10

2.3 Ethereum 11

2.3.1 Gas 11

2.4 Tendermint and the Cosmos SDK 11

2.5 Smart Contract Languages 12

2.5.1 Solidity 13

2.5.2 Vyper 14

2.5.3 LIGO 15

2.5.4 Others 16

3 Related Work **17**

3.1 Evaluation of Logic-based Smart Contracts for Blockchain Systems (2016) 17

3.2 Blockchain and Beyond: Proactive Logic Smart Contracts (2018) 19

3.3 Towards Declarative Smart Contracts (2020) 21

3.4 A Method of Logic-based Smart Contracts for Blockchain Systems (2018) 21

3.5 Logic-based Smart Contracts (2020) 22

3.6 Others 23

3.7 Summary 24

4 Chainlog **25**
4.1 Requirements . . . 25
4.1.1 Language Requirements . . . 25
4.1.2 Blockchain Requirements . . . 27
4.2 Language Design . . . 28
4.2.1 Syntax . . . 28
4.2.2 Semantics . . . 29
4.3 Blockchain Specification . . . 35
4.3.1 Accounts, Addresses and Authentication . . . 35
4.3.2 Queries . . . 36
4.3.3 Transactions . . . 36
4.3.4 Blocks . . . 37
4.3.5 State Machine Specification . . . 37
4.4 Summary and Extensions . . . 41

5 Implementation **42**
5.1 Language Implementation . . . 42
5.1.1 Core Interpreter . . . 42
5.1.2 Go-Chainlog . . . 44
5.1.3 Other Tools . . . 45
5.2 Blockchain Implementation . . . 46
5.2.1 Architecture . . . 46
5.2.2 Transaction Lifecycle . . . 48
5.2.3 Query Lifecycle . . . 49
5.2.4 Protocol Buffers . . . 50
5.2.5 Contract Representation . . . 50
5.2.6 Contract Execution . . . 51
5.3 Summary and Extensions . . . 52

6 Evaluation **53**
6.1 Demonstration of the Language . . . 53
6.1.1 Crowdfunding . . . 53
6.1.2 Licence . . . 56
6.1.3 Parametric Insurance . . . 58
6.1.4 Pharmacogenomics Data . . . 63
6.2 Security . . . 67
6.2.1 Denial of Service Protection . . . 67

6.2.2 Determinism . . . 68
6.2.3 Code Vulnerability Mitigation . . . 68
6.2.4 User-level Concerns . . . 69
6.2.5 Other Security Considerations . . . 69
6.3 Summary and Extensions . . . 69

7 Conclusion **71**
7.1 Future Work . . . 71
7.2 Ethical and Legal Discussion . . . 72
7.3 Final Remarks . . . 72

A Full Code Listings **78**
A.1 Crowdfunding . . . 78
A.1.1 Chainlog . . . 78
A.1.2 Solidity – Simple . . . 78
A.1.3 Solidity – Extended . . . 79
A.2 Licence . . . 81
A.2.1 Chainlog . . . 81
A.2.2 Solidity . . . 81
A.3 Parametric Insurance . . . 83
A.3.1 Chainlog . . . 83
A.3.2 Solidity . . . 84
A.4 Pharmacogenomics Data . . . 86
A.4.1 Chainlog . . . 86
A.4.2 Solidity . . . 86

Chapter 1

Introduction

Since the emergence of Bitcoin and Ethereum, much interest has been centred around blockchain-enabled *smart contracts*: programs that are able to encode agreements and carry out transactions without the need for a trusted intermediary. Transparency, security and automation are some of the benefits they promise. Today, blockchain-based smart contracts have applications spanning financial services, Internet of Things, supply chain management, real estate and more [1].

Smart contracts are typically written in imperative programming languages. The most prevalent is Solidity [2], a C++ and JavaScript inspired curly-bracket language designed specifically for writing contracts targeting the Ethereum [3] platform. Nonetheless, the landscape of languages has seen the introduction and fall to obscurity of numerous alternatives identified in the literature [4], including a handful of functional programming languages.

Far less consideration has been devoted to the logic programming paradigm, however. Given the rule-based, condition-consequent nature of contractual clauses, the Horn clause rules of logic-based languages may present a compelling formalism. Consider, for example, an earthquake insurance contract, which pays out in percentages based on the highest recorded seismic intensity. Listing 1.1 presents a conceptualisation in a logic programming style.

```
1 payout_percentage(100) :- seismic_intensity(Intensity),
2                           (Intensity = '6+' or
3                            Intensity = '7').
4 payout_percentage(80)  :- seismic_intensity(Intensity), Intensity = '6-'.
5 payout_percentage(40)  :- seismic_intensity(Intensity), Intensity = '5+'.
```

Listing 1.1: Earthquake Insurance in Logic

In a logic programming language, the programmer declares a set of definitions in the form of logical implications. The first rule, for example, reads: "the payout percentage is 100% if there was a recorded seismic intensity of 6+ or 7". This is in contrast to the equivalent Solidity expression given in listing 1.2, where the programmer is responsible for delineating the step-by-step procedure to set the `payoutPercentage` variable to its appropriate value. Logic-based languages constitute a high-level, executable specification, with a reading that mirrors natural language. They allow the programmer to specify *what* the contractual terms are, without needing to detail *how* to accomplish them. The result is a more readable and comprehensible representation. In addition, it mitigates opportunities for the programmer to make errors [5].

The purpose of this work is to investigate the application of a logic-based language to blockchain-based smart contracts. Our objective is to empower the developer with a framework for writing, deploying and executing smart contracts as logic programs. In doing so, we enable parties to engage in computerised agreements that retain a understandable, specification-like representation, are more error-free, and are regulated by a trustless, transparent and tamper-free blockchain system.

```solidity
uint8 public payoutPercentage;

function evaluatePayoutPercentage(string calldata intensity) external {
    bytes32 _intensity = keccak256(abi.encodePacked(intensity));
    if (_intensity == keccak256(abi.encodePacked("6+"))
        || _intensity == keccak256(abi.encodePacked("7"))) {
        payoutPercentage = 100;
    } else if (_intensity == keccak256(abi.encodePacked("6-"))) {
        if (payoutPercentage < 80) {
            payoutPercentage = 80;
        }
    } else if (_intensity == keccak256(abi.encodePacked("5+"))) {
        if (payoutPercentage < 40) {
            payoutPercentage = 40;
        }
    }
}
```

Listing 1.2: Earthquake Insurance in Solidity

1.1 Contributions

We introduce **Chainlog**, a new language and blockchain to support logic-based smart contracts. Our contributions are categorised into two artefacts:

- **Design and Implementation of the Chainlog Language**
  We present Chainlog, a domain-specific, hybrid logic-based and imperative language. A definition of the language's syntax and semantics is given in section 4.2. We give a working implementation written in Prolog and Go, described in section 5.1. Our showcase in chapter 6 demonstrates the ability of the language to express a broad range of contracts. Compared to Solidity, we find our language to be close to specification and identify an array of errors and vulnerabilities which Chainlog mitigates. Additionally, we illustrate Chainlog's powerful and flexible querying capabilities.
- **Design and Implementation of the Chainlog Platform**
  We present the Chainlog Platform, a complete blockchain system built from the ground up to facilitate the deployment, querying and execution of smart contracts written in the Chainlog language. The platform features a native cryptocurrency which endows contracts with the ability to induce monetary transfers. A formal specification of the blockchain is provided in section 4.3. We describe our operational implementation in section 5.2. Our security evaluation in section 6.2 demonstrates its robustness and highlights properties of decentralisation, transparency, inspectability and immutability.

Chapter 2

Background

2.1 Blockchain

A blockchain is a replicated, transaction-based state machine [3]. At any given time, the machine is in a global, canonical *application state* σ. A change in state is triggered by an external input called a *transaction*, written T. The machine's *state transition function* written *apply* defines how a transaction T should update the current state σ, so as to obtain a new state σ':

$$\sigma' = apply(\sigma, T)$$

A blockchain system starts with some initial state called the *genesis state* $\sigma_{genesis}$. Its operation involves an infinite process of accepting incoming transactions, executing the transition function and updating the canonical state accordingly.

To create a blockchain, one must therefore define what the state of the system σ comprises and what the transition function *apply* does.

Let us reify this with an example. Imagine a blockchain which implements a simple cryptocurrency, which we shall call "SimpleCoin". A person may possess a certain number of SimpleCoins. In addition, a person may at any time send a number of their SimpleCoins to another person. For simplicity, suppose that Alice and Bob are the only two users of SimpleCoin and that they both currently hold 100 SimpleCoins each.

The state of the blockchain σ needs to represent how many SimpleCoins each person possesses. A suitable representation of σ would therefore be a mapping from people to their SimpleCoin balances. Our current canonical state is the following:

$$\sigma = \{Alice \mapsto 100, Bob \mapsto 100\}$$

Suppose Alice wishes to send 30 of her SimpleCoins to Bob. Evidently, this transfer requires a change to be made to the state. It therefore requires a transaction to be input to the SimpleCoin state machine. We can represent this transaction as an object $T = send(Alice, Bob, 30)$.

The SimpleCoin blockchain's state transition function *apply* is responsible for computing the new set of balances σ' that should result from the transaction T. In general, given a transaction of the form $send(From, To, Amount)$, we would expect the implementation of this transition function to subtract $Amount$ from the balance of $From$ and add it to the balance of To. For our example transaction T, this would result in the new state σ' shown below:

$$\begin{aligned}\sigma' &= apply(\{Alice \mapsto 100, Bob \mapsto 100\},\ send(Alice, Bob, 30)) \\ &= \{Alice \mapsto 70, Bob \mapsto 130\}\end{aligned}$$

Thus, when Alice sends her transaction T to the SimpleCoin blockchain, it causes the canonical state to transition from σ to this new σ', thereby effecting the transfer.

2.1.1 Replication and Decentralisation

It would be straightforward to implement a blockchain's state machine in a centralised manner: a single server would store the state on disk and service incoming transactions to transition it. The issue with this arrangement is that the canonical state is controlled entirely by this single entity; users must submit their trust that the centralised server will not fail or misbehave. Instead, the canonical state of a blockchain system is an abstract notion maintained by a distributed network of nodes.

Each node is a replica of the blockchain's state machine: it has a local copy of the state σ and an implementation of the state transition function *apply*. The nodes are in constant peer-to-peer communication with each other to maintain consensus on what the current state should be. It is this agreed state that constitutes the global, canonical σ.

In order to transition the canonical state, a transaction must be broadcast across the entire network and processed by every node's local state machine. Each node will independently execute the transition function on their copy of σ and compute the new state σ'. Importantly, they will all arrive at the exact same σ'. After coming to consensus on this new state, the globally-agreed canonical state becomes σ', and the effects of the transaction become actualised.

For the above to work, it is vital that the transition function *apply* is *deterministic*, so that there is no contention on what the new state should be. That is, there should be one and only one possible new state σ' for a given input state σ and transaction T. This criterion will become important in the later chapters of this work.

Because the canonical state is a consolidation of many individual, local replicas, it achieves the advantage of being *decentralised*. No single entity is in control of it. If a node fails or attempts to alter the state maliciously, it falls out of accord with its peers; the remainder of the network continues to agree on the correct state.

2.1.2 Blocks

In reality, a blockchain system's canonical state is not progressed by each individual transaction. A number of transactions are batched into a data structure called a *block*, and the entire block is committed at once. This disperses updates to the global state in order to give the network time to reach agreement [6].

The list of transactions included in a block forms its *body*. A block also has a *header* containing metadata. While the exact contents of the header differ between blockchain systems, they all typically feature a number of core fields:

- **Block number/height**: an incremental number. The first ("genesis") block has a height of zero and each subsequent block has a height of one plus that of the block that came before it.
- **Timestamp**: the timestamp of the block's creation.
- **Hash of Transactions**: the Merkle root hash [7] of the transactions in the body. This allows transaction membership to be confirmed efficiently.
- **Hash of Previous Block**: the hash of the header of the previous block. This uniquely identifies the predecessor and can therefore be thought of as a pointer to it.

The implication of the last field is that the entire history of transactions is maintained in one long chain. One is able to start from the latest block and follow the hash pointers back to the genesis block. Hence, the "blockchain".

Observe that the block makes no mention of the current canonical state σ. This σ exists only at the application level; it is not actually stored in the blockchain data structure. Because the state transition function *apply* is deterministic, it is always possible to recover the current σ by starting at $\sigma_{genesis}$ and applying every transaction from every block in the chain in sequence. Despite this technicality, the chapters of this work (and most other work) may forgo this distinction and refer to the contents of σ as being "on the blockchain" or simply "on-chain".

At any time, the hash of the latest block's header can serve to verify the current state of σ. Consider again our SimpleCoin blockchain where Alice transferred 30 of her SimpleCoins to Bob. Suppose that at some later date, Bob conspires with one of the blockchain's nodes to give himself more SimpleCoins, by altering Alice's *send* transaction to increase the amount to 100. This attempt will fail. Modifying Alice's transaction will cause the hash of the transaction to change. This will in turn cause the hash of transactions field in the enclosing block's header to change, which will cause the hash of the entire header to change. The hash of the header is included as a field in the next block; so, the hash of the next block's header will also change, and the block after that, etc. This propagates to the hash of the latest block's header, which will be found to disagree with the rest of the network. The consequence is that the blockchain is completely *immutable*.

2.1.3 Transactions and Queries

There are two ways an end user interacts with a blockchain: *transactions* and *queries*.

We have seen that transactions are objects which trigger state transitions. It is useful to regard them with the database sense of the word, as several of the properties of database transactions apply. An important one is atomicity: a transaction must execute in its entirety or not at all. Almost all blockchain transactions in practice have some notion of failure or invalidity; in such cases the transaction should be aborted and no updates to state should result. To use the SimpleCoin example, an attempt by Alice to send 200 coins to Bob should be deemed invalid because Alice lacks the sufficient balance. The transaction should enact no changes to state; we should never observe a partial execution where e.g. Bob's balance increased by 200 but Alice's balance remained unchanged. In the more complex state transition functions of this work, we will see that the throwing of exceptions becomes an important method of signalling that a transaction should be aborted and state reverted.

Queries are important for *transparency*. They allow users to request information about the blockchain application state σ and the state of the network in general. For example, it would be sensible for the SimpleCoin blockchain to support the querying of a person's balance.

Unlike transactions, queries cannot make updates to state. They are therefore able to be serviced by a single node reading from its local copy of σ. There is no need to broadcast a query to the entire network and no need to attain consensus. This makes them comparatively cheaper. However, it does not prevent them from invoking arbitrarily complex computation – anything that can be evaluated in-memory is potentially viable.

2.1.4 Consensus

Consensus is the process by which nodes reliably and securely come to agreement on the state of the blockchain. Current systems predominantly employ one of two schemes: Proof-of-Work (PoW) or Proof-of-Stake (PoS) [8]. These protocols are not only used to establish consensus; they are also responsible for assigning one node to be the creator of each new block, who will be granted a monetary reward – an incentive for nodes to participate in the network.

Under Proof-of-Work, *mining* is the process of aggregating transactions into blocks and proposing their inclusion to the blockchain. The nodes which perform this are called *miners*. A miner will select pending transactions to include in a new block, and is responsible for verifying the transactions are valid. It must then solve a computationally expensive cryptographic problem in order for the block to be accepted. The cryptographic puzzle is such that the answer is easily verifiable and serves as certificate of the fact that the expensive work has been done. On completion, the block and the certificate are broadcast to the rest of the network. The other nodes verify the certificate and append the new block to their local copy of the blockchain. Consensus is reached when at least 51% of nodes agree on the new global state of the network.

Miners are in constant competition to solve a block's cryptographic problem, as the first solver is granted the status of block creator and the associated reward. The process of mining is extremely energy intensive and requires dedicated hardware to be profitable [9]. This scheme makes it near

impossible for a malicious miner to create a block including fake transactions – they would need to solve the block's cryptographic problem faster than every other miner in the network.

Proof-of-Stake emerged as a more energy efficient alternative to Proof-of-Work [10]. The creation of blocks is performed by *validators*: nodes which have staked an amount of cryptocurrency into a security deposit as collateral. A block's creator is elected at random from the set of validators, weighted by the amount they have staked. As in PoW, the creator will broadcast the new block to the network and earn a reward for their effort.

In Proof-of-Stake, malicious behaviour is prevented by economic disincentives: a validator that misbehaves will have some of their stake forfeited. This is called "slashing".

2.2 Smart Contracts

A smart contract is a computer protocol that automatically enforces the contractual terms of an agreement [11]. The term "smart contract" was introduced by Nick Szabo in the 1990s [12] but became popularised with the introduction of Ethereum nearly twenty years later [3]. It is today used to refer to general purpose programs that run on the blockchain.

Smart contracts can be thought of as computer programs whose code is stored as part of a supporting blockchain's state σ. These programs consist of logic following the form of "when called with X, do Y", i.e. they are reactive. The blockchain allows users to send transactions of the form "call contract C with X", thereby triggering the execution of Y. More specifically, the blockchain's transition function *apply* knows to interpret this transaction by fetching and executing the code of C, passing it X. Then, the code of C reacts to X by performing Y. The Y of these programs can involve complex conditions, enabling the encoding of arbitrary contractual terms. Furthermore, they have the ability to send and receive money in the form of the blockchain's native cryptocurrency.

Because smart contracts are stored in the application state σ and executed as part of the transition function *apply*, they are stored and executed on every node in the network. Hence, they inherit the advantages of blockchain technology described in the previous section. They are decentralised: their operation is not controlled by any one party, and so they are trustless. They are immutable: once a contract is formalised, the terms cannot be tampered with. Assuming the blockchain provides functionality to query the programs' code and state, they are also transparent: the terms are available for all parties to view at any time.

These benefits have made the encoding of traditional paper contracts as blockchain-based smart contacts appealing. Currently, smart contracts have applications in finance, real estate, insurance, supply chain management, healthcare, gambling and more. We will explore many of these use cases in the chapters that follow.

2.2.1 Determinism

In order for blockchain nodes to come to consensus, smart contract execution must be fully deterministic. To adhere to this property, it is necessary for contracts to execute in complete isolation: they must have no access to the filesystem, the network or other processes running on the node, as these are potential sources of non-determinism.

This may appear to be a severe limitation. Many contracts depend on information from the outside world. For example, a manufacturing and supply agreement may stipulate the supplier to be in default if the products are not delivered by a certain date. A smart contract representing this agreement needs to know on what date the products are delivered.

The only way to endow a contract with external knowledge is to upload the information to the blockchain. The buyer in the above example could manually inform the contract when the products are received. Alternatively, we might conceive this to be automated by IoT smart sensor system which detects when the products are delivered.

2.3 Ethereum

Ethereum is the most mature blockchain-based smart contract platform [13]. It supports the deployment and execution of contracts as a Turing-complete bytecode referred to as Ethereum Virtual Machine (EVM) code. In addition, Ethereum provides a native cryptocurrency called ether (ETH), which smart contracts have the ability to send and receive. As of writing, Ethereum adopts a Proof-of-Work consensus protocol, but is due to move to Proof-of-Stake in the future.

The Ethereum ecosystem consists of accounts, which can belong to external users or smart contracts. Every account is identified by a unique address. The state σ of Ethereum represents the state of all accounts: it is a mapping from addresses to so-called account states, structures containing four fields: an ether balance, a nonce (used to prevent replay attacks), a hash of smart contract code and a Merkle root of smart contract storage (essentially the state of a contract's global variables) [14]. The latter two fields are empty for external user accounts.

An Ethereum transaction is one of two types: a contract creation or a message call. The former is used to deploy new contracts to the platform. The latter has dual purpose, used for simple transfers of ether between external users and for invoking smart contract execution.

Ethereum's state transition function Υ is quite complex and is not covered here. Its full specification can be found in the yellow paper [14]. In this work, we will only discuss parts of its operation as they become relevant.

Although smart contracts are deployed, stored and executed on the Ethereum platform as EVM code, developers usually write their contracts in a high-level language like Solidity [2], which is compiled down to bytecode. Solidity and other languages will be explored in section 2.5.

2.3.1 Gas

Because the EVM bytecode has the power to express loops, smart contract execution may in theory never terminate. This is problematic: if one were to invoke a contract with an infinite loop, all the nodes on the Ethereum network would become stuck executing it, leading to a denial of service. To address this, Ethereum employs a pricing mechanism. All smart contract computation incurs a fee of gas, which is paid in ether.

When a user sends a transaction, they must specify an upper bound on the amount of gas it will be allowed to consume. This is effectively an upfront payment. Throughout a smart contract's execution, gas is gradually depleted based on the amount of computational work done. If the upfront payment is insufficient to cover the cost of the computation, an out-of-gas (OOG) exception is immediately thrown, and all changes to state are reverted. A transaction therefore specifies a finite limit on the amount of computation it can spawn.

Note that this scheme only applies to transactions. It is not necessary to charge a fee for queries, since they only need to be executed on one node.

2.4 Tendermint and the Cosmos SDK

Tendermint is a general purpose blockchain engine for replicating applications across many machines [15]. It differs from systems like Ethereum in that it decouples the consensus engine and low-level networking from the application running atop it. Developers are thus able to build arbitrary applications written in any programming language, and rely on Tendermint to replicate it securely and consistently.

There are two fundamental components to the Tendermint software: the Tendermint Core and the Application Blockchain Interface (ABCI). Tendermint Core is the consensus engine, responsible for ensuring consistent state across the network. It designates a Byzantine Fault Tolerant (BFT) Proof-of-Stake protocol, the former qualifier meaning it can tolerate up to $\frac{1}{3}$ of validators failing arbitrarily or becoming malicious. The ABCI is the API which defines how the consensus engine

communicates with the application process. A developer connects their application to Tendermint by implementing this interface. Tendermint relays information to the application through a set of defined ABCI methods each indicating the occurrence of a certain event, e.g. the receipt of a transaction or the creation of a new block. It is then up to the application to interpret this information.

The Cosmos SDK is an open-source framework for building custom, application-specific blockchains from scratch [16]. It is an abstraction above Tendermint, providing a boilerplate implementation of the ABCI. In addition, it offers facilities to take care of some of the common blockchain application needs such as cryptographic signature verification.

In effect, the Cosmos SDK and Tendermint allow one to define their own state machine σ and *apply*.

The Cosmos SDK imposes some design decisions on the applications built with it: notably, a modular architecture and an object-capability approach to state management.

Applications built with the Cosmos SDK consist of a collection of *modules*. Each module should correspond to one application concern. It manages a partition of the application state σ and defines a set of transaction types which it processes. Thus, a module can be seen as a smaller state machine within the large state machine of the whole application. The advantage of modules is composability: it is easy to integrate new modules into an application or reuse a module in multiple applications. In fact, the Cosmos SDK ships with a number of core modules that implement common blockchain functionality. For example, the `bank` module manages currency – its partition of state is used to store account balances, and it offers methods for sending money from one account to another.

The modular design makes it easy for developers to incorporate functionality already written by other people, so they do not have to implement it themselves. However, there arises the possibility for a third-party module to contain vulnerable or malicious code. This is the motivation for the Cosmos SDK's object-capability-based security. Each module in a Cosmos application has one or more *keepers* which control access to its partition of the state; all reads and writes to the module's state must go through one of its keepers. If module A requires access to module B's state, module A must be given a reference to one of module B's keepers. Module A's interaction with the state of module B is restricted to the methods exposed by that keeper. Object-capability follows the principle of least privilege: module B's keeper should only provide the minimum access necessary for module A to complete its task.

2.5 Smart Contract Languages

As blockchain-based smart contracts are a relatively recent emergence, the landscape of languages for writing smart contracts is not as well-developed as that of traditional programming languages. Even the better-established among them are still very young and in rapid development [17]. Research on smart contract languages is seeing the proposal of new languages as fast as their abandonment, but options for a well-maintained and well-documented smart contract language remain limited.

The primary purpose of this section is to introduce the leading option, Solidity, which is the default choice of language for Ethereum development. It will serve as the main point of comparison for the language of this work.

Several other languages were investigated during the early research of this project. Although far more obscure, two of these are presented in this section to provide a flavour for the alternatives to Solidity. We introduce Vyper, the second most popular language for Ethereum contract development [18]. We also illustrate LIGO as an example of a functional language.

2.5.1 Solidity

Solidity [2] is the most widely used language for smart contract programming [19]. Object-oriented and statically typed, the language takes inspiration from JavaScript and C++ and was designed specifically to express contracts. Listing 2.1 presents an example of an open auction smart contract expressed in Solidity, adapted from the official documentation.

```
pragma solidity ^0.8.4;

contract SimpleAuction {
    address payable public beneficiary;
    uint public auctionEndTime;

    address public highestBidder;
    uint public highestBid;

    mapping(address => uint) pendingReturns;

    bool ended;

    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    error AuctionAlreadyEnded();
    error BidNotHighEnough(uint highestBid);
    error AuctionNotYetEnded();
    error AuctionEndAlreadyCalled();

    constructor(
        uint biddingTime,
        address payable beneficiaryAddress
    ) {
        beneficiary = beneficiaryAddress;
        auctionEndTime = block.timestamp + biddingTime;
    }

    function bid() external payable {
        if (block.timestamp > auctionEndTime)
            revert AuctionAlreadyEnded();

        if (msg.value <= highestBid)
            revert BidNotHighEnough(highestBid);

        if (highestBid != 0) {
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    function withdraw() external returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            pendingReturns[msg.sender] = 0;

            if (!payable(msg.sender).send(amount)) {
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    function auctionEnd() external {
        if (block.timestamp < auctionEndTime)
            revert AuctionNotYetEnded();
        if (ended)
            revert AuctionEndAlreadyCalled();

        ended = true;
        emit AuctionEnded(highestBidder, highestBid);
```

```
66
67         beneficiary.transfer(highestBid);
68     }
69 }
```

Listing 2.1: Open Auction Smart Contract in Solidity [20]

A Solidity contract is somewhat similar to a class in an object-oriented programming language, consisting of a series of state variables and functions, and even supporting inheritance. Functions are the way in which users trigger computation: they can be called from outside the blockchain by an Ethereum message call transaction provided they are given `external` visibility. Functions declared `payable` allow the user to send ether to the contract with the message call.

Variables in Solidity can reside in one of three data locations: storage, memory and calldata. Storage variables exist as part of the contract's account state and persist between message calls. The state variables are stored here. Memory variables include those declared inside a function; they exist for the duration of an external function call. Finally, calldata is a special read-only location where the arguments to an external function are stored.

In addition to standard primitive types such as `uint` and `bool`, Solidity features an `address` type to represent the address of an Ethereum account. The language also supports structs, arrays and mappings, although their functionality is more limited then their equivalents in other languages (e.g., mappings cannot be iterated).

If a function does not modify storage, Solidity allows it to be declared with the `view` keyword. If it further does not read from storage, it may be declared `pure`. Functions of both these types possess a special quality: they can be invoked by a blockchain query. This is interesting because it enables some computation to be performed without incurring a gas fee.

Although Solidity presents itself as an object-oriented language, it is in reality still quite low-level. Inline assembly is not uncommon to see, and is a source of dread for many developers [21]. One often finds oneself working with raw bytes, as we will discover in the evaluation chapter. As Solidity is still a relatively new language, the compiler is rapidly evolving. This has led to frustration for developers, as new updates to the language are not backward-compatible [22].

Solidity has received criticism due to inherent vulnerabilities in the language's design [1, 23]. This has led to the creation of numerous tools for the analysis of smart contract code such as Mythril [24], Slither [25], Manticore [26], Oyente [27] and many others [28, 29]. It is lamentable that the language warrants the need for such tools.

2.5.2 Vyper

Vyper [30] is a Pythonic smart contract language with a focus on security, auditability and clarity. The language intentionally restricts the expression of certain code constructs deemed to hinder these goals, including inheritance, infinite-length loops and operator overloading. It attempts to address the vulnerabilities plaguing Solidity, to some success [31]. The example of the open auction smart contract written in Vyper is shown in listing 2.2.

```
1  beneficiary: public(address)
2  auctionStart: public(uint256)
3  auctionEnd: public(uint256)
4
5  highestBidder: public(address)
6  highestBid: public(uint256)
7
8  ended: public(bool)
9
10 pendingReturns: public(HashMap[address, uint256])
11
12 @external
13 def __init__(_beneficiary: address, _auction_start: uint256,
14              _bidding_time: uint256):
15     self.beneficiary = _beneficiary
16     self.auctionStart = _auction_start
```

```
17     self.auctionEnd = self.auctionStart + _bidding_time
18     assert block.timestamp < self.auctionEnd
19 
20 @external
21 @payable
22 def bid():
23     assert block.timestamp >= self.auctionStart
24     assert block.timestamp < self.auctionEnd
25     assert msg.value > self.highestBid
26     self.pendingReturns[self.highestBidder] += self.highestBid
27     self.highestBidder = msg.sender
28     self.highestBid = msg.value
29 
30 @external
31 def withdraw():
32     pending_amount: uint256 = self.pendingReturns[msg.sender]
33     self.pendingReturns[msg.sender] = 0
34     send(msg.sender, pending_amount)
35 
36 @external
37 def endAuction():
38     assert block.timestamp >= self.auctionEnd
39     assert not self.ended
40 
41     self.ended = True
42 
43     send(self.beneficiary, self.highestBid)
```

Listing 2.2: Open Auction Smart Contract in Vyper [32]

Aside from its Python-inspired syntax, Vyper contracts are similar to their Solidity counterparts in structure and style. They consist of state variables and functions, have the same basic keywords (`external`, `payable`, `view`, `pure`, etc.), support the same types (addresses, structs, arrays, maps) and follow an imperative style.

2.5.3 LIGO

To give an idea of the declarative languages for smart contract development, we present in listing 2.3 a crowdfunding contract expressed in LIGO [33], which was written as part of this work. LIGO is designed for writing contracts for the Tezos blockchain [34], but the principles are the same as those of Ethereum. The language actually offers four different flavours of syntax; the one shown here is the OCaml inspired variant, CameLIGO.

```
1  type funder = {
2    sender : address;
3    amount_funded : tez
4  }
5 
6  type parameter = Fund | Withdraw
7 
8  type storage = {
9    funders : (nat, funder) big_map;
10   next_funder_index : nat
11 }
12 
13 type return = operation list * storage
14 
15 let owner : address = ("tz1TGu6TN5GSez2ndXXeDX6LgUDvLzPLqgYV" : address)
16 
17 (* Initial storage *)
18 let init_storage : storage = {
19   funders = (Big_map.empty : (nat, funder) big_map);
20   next_funder_index = 0n
21 }
22 
23 let fund (storage : storage) : return =
24   (* Add new entry to funders big_map *)
25   let funders = Big_map.update
26     storage.next_funder_index
```

```
27      (Some { sender = Tezos.sender ; amount_funded = Tezos.amount })
28      storage.funders
29    in
30    (([] : operation list), { funders = funders ; next_funder_index = storage.next_funder_index + 1n
        })
31
32  let withdraw (storage : storage) : return =
33    if Tezos.sender <> owner then
34      failwith "Caller is not the owner"
35    else
36
37    (* Get owner's account as unit contract so that funds may be transferred to it *)
38    let receiver : unit contract =
39    match (Tezos.get_contract_opt owner : unit contract option) with
40    | Some contract -> contract
41    | None -> failwith "No contract"
42    in
43
44    (* Define transfer operation *)
45    let withdrawal : operation = Tezos.transaction unit Tezos.balance receiver in
46
47    (([withdrawal] : operation list), storage)
48
49  (* Entrypoint *)
50  let main (action, storage : parameter * storage) : return =
51    match action with
52    | Fund ->     fund storage
53    | Withdraw -> withdraw storage
```

Listing 2.3: Crowdfunding Contract in CameLIGO

When an external user calls the contract, they supply an action to be performed – in this case, either `Fund` or `Withdraw`. Execution starts at the entrypoint function `main` found at the bottom of the code. Where the functions of Solidity and Vyper issued imperative commands to write to storage, perform transactions, etc, the entrypoint of CameLIGO is instead written in a functional programming style. It takes as input the action and current state of storage, then returns the new storage and a list of external operations to be performed. There are no side-effects – the function defines what the new state is in terms of the action and old state.

The functional approach means LIGO contracts lend themselves to formal verification [33]. However, the program arguably still "looks imperative". Error checking is exception-based through the error-raising `failwith` function. Operations such as the transfer of cryptocurrency are still very much procedural instructions; the only difference is that LIGO functions return them in a list to be performed later.

2.5.4 Others

While the smart contract languages in widespread use are small in number, the list of new languages being proposed in the literature is ever growing. We give a brief mention to some of the languages which are still in the implementation phase or are no longer maintained.

Obsidian [35] is described as a state-oriented language which models smart contracts as explicit finite state machines. It is designed to make the development of contracts easier and more bug-free. Currently, the language supports the Hyperledger Fabric platform [36], but is still in early development.

Bamboo [37] is similar in that it makes the state of the contract explicit. Functions must declare what state the contract should move to when returning. It is otherwise a high-level, imperative language similar to Solidity. Bamboo has had no updates since 2018.

Flint [38] is a statically-typed, imperative language with a focus on safety. It introduces novel constructs for authorisation and asset handling. The language is no longer maintained.

For more languages, the reader is directed to a literature review such as [4].

Chapter 3

Related Work

The domain of smart contract languages has yet to see a logic-based language in use. Nevertheless, the idea has been pondered in the literature, and there have been a handful of attempts to realise an implementation to support it. In this chapter, we critically examine the existing work in this area. Our analysis has two foci: the first regards the expressiveness of the logic languages of choice and their suitability in the blockchain context; the second concerns the feasibility of the implementation options.

3.1 Evaluation of Logic-based Smart Contracts for Blockchain Systems (2016)

Governatori et al. were the first to consider logic-based smart contracts in their 2016 paper [5]. The study investigates the advantages and challenges of the logic paradigm from a legal and technical perspective. They argue that existing procedural approaches are ill-suited for encoding legal reasoning and stand to benefit from the incorporation of logic-based languages, which are easier to understand, easier to validate and more compact. A follow up by the authors in 2018 [39] reiterates much of the same sentiments. We analyse the content of both works in this section.

Governatori et al. propose the use of defeasible logic for the expression of contracts. This is motivated by an example of a product evaluation licence, the terms of which are given in example 3.1.

Article 1 *The Licensor grants the Licensee a license to evaluate the Product.*

Article 2 *The Licensee must not publish the results of the evaluation of the Product without the approval of the Licensor; the approval must be obtained before the publication. If the Licensee publishes results of the evaluation of the Product without approval from the Licensor, the Licensee has 24 hours to remove the material.*

Article 3 *The Licensee must not publish comments on the evaluation of the Product, unless the Licensee is permitted to publish the results of the evaluation.*

Article 4 *If the Licensee is commissioned to perform an independent evaluation of the Product, then the Licensee has the obligation to publish the evaluation results.*

Article 5 *This license terminates automatically if the Licensee breaches this Agreement.*

Example 3.1: Product Evaluation Licence [5]

Listing 3.1 gives the authors' representation of the licence in Formal Contract Logic (FCL), a deontic defeasible logic. It is designed to be evaluated by a defeasible logic engine. The purpose of the code is to evaluate the permissions, obligations and prohibitions of the licensee to use the product, publish evaluation results, publish comments and remove published material (`use`, `publish`, `comment`, `remove` respectively), based on whether or not the licensee has been granted the licence,

granted approval to publish and commissioned to perform an independent evaluation (`hasLicense`, `hasApproval`, `isCommissioned` respectively). Article 4 of the licence for example obligates the licensee to publish the results if commissioned; this is encoded by the rule on line 7 dictating that the obligation to publish is true if it is the case that `hasLicense` and `isCommissioned` are both true.

```
Art1.0: => [Forb_licensee] use
Art1.1: hasLicense => [Perm_licensee] use
Art2.1: => [Forb_licensee] publish [Compensated] [Obl_licensee] remove
Art2.2: hasLicense, hasApproval => [Perm_licensee] publish
Art3.1: => [Forb_licensee] comment
Art3.2: [Perm_licensee] publish => [Perm_licensee] comment
Art4.0: hasLicense, isCommissioned => [Obl_licensee] publish
Art5.1: violation => [Forb_licensee] use
Art5.2: violation => [Forb_licensee] publish
% Superiority relation
Art1.1 > Art1.0,
Art2.2 > Art2.1, Art3.2 > Art3.1,
Art5.1 > Art1.1, Art5.2 > Art4.0.
```

Listing 3.1: FCL Licence Contract [5]

The advantage of the authors' logic formalism is that the terms of the contract are able to be written as high-level rules. The programmer specifies what the legal reasoning of the contract is, but does not need to specify the step-by-step computations to achieve it, as they would in an imperative language. Governatori et al. suggest this results in more error-free contracts.

Use of deontic logic bears reification of the normative concepts of permission, obligation and prohibition as constructs in the language. This means the evaluation of these norms is handled by the reasoning engine. For example, when a permission becomes true, the engine determines that the prohibition should be false; the programmer does not have to set it manually. While deontic logic has been shown to be effective at capturing legal reasoning, it constrains the scope of the language to the contracts that operate on its normative statuses. Many contracts, such as business and financial contracts, are not naturally expressed by notions of obligation and prohibition. Ideally, we would like a language that is more general. We would like a class of logic where contractual terms are still expressed as logical implications – so as to realise the aforementioned benefits of high-level and error-free programs – but possesses constructs which can express a broader range of contracts.

The FCL representation of the contract is also limited in the sense that it only determines the normative consequences; it cannot enforce them. Governatori et al. recognise this. They highlight that the declarative and imperative paradigms are not incompatible, and allude to the prospect of a hybrid approach combining logic-based and procedural elements. The logic-based component establishes the effects, while the procedural component performs the actions needed to implement them.

Both the 2016 and 2018 papers are exploratory – there is no implementation given. Nevertheless, the authors discuss a number of possible methods for realising logic-based smart contracts on existing blockchain systems. The challenge surrounds the inference engine, which gives rise to two main approaches:

- Off-chain inference, i.e. inferences are made by a separate server outside the blockchain system. Because smart contracts cannot interact with the outside world, an on-chain smart contract is not able to call an off-chain inference engine. The authors instead suggest to execute contracts off-chain and only store the knowledge bases on-chain. The off-chain inference mechanism will update the blockchain with its conclusions by calling to a procedural smart contract on-chain. This procedural element will then carry out any transactions based on the inferential conclusions.

- On-chain inference, i.e. inference is performed by the nodes of the blockchain. Although smart contracts on the blockchain cannot make outside calls, they are able to interact with other contracts. Thus, it is proposed to write the inference engine into a generic, meta-program smart contract. Other contracts can call into this, passing their particular knowledge base and rules.

Both approaches have drawbacks. The problem with the off-chain method is trust. When computation is moved off-chain, benefits of the blockchain such as security and verifiability are lost. A trusted third-party inference service is required. The on-chain method does not suffer from this issue as the inference mechanism is implemented as a transparent, immutable contract on the blockchain. However, the implementation of meta-programs and the encoding of programs as data would be unwieldy within the rigid structures of existing systems. In addition, performing this computation on-chain would be costly.

Governatori et al. brought to light the merits of logic-based languages for representing smart contracts. Their legal focus led to the suggestion of deontic defeasible logic, which we lamented as being too specialised. In addition, we have seen the need to address several challenges in order to support a logic language, such as how to enforce contractual terms and how to integrate a logic engine into a blockchain system. Perhaps the most compelling proposition is for the coalescence of imperative and declarative approaches, which could realise the best of both paradigms: the aptitude of logic programming to express rules and of procedural programming to enact commands.

3.2 Blockchain and Beyond: Proactive Logic Smart Contracts (2018)

Maffi [40] presents a proof-of-concept for a system supporting proactive smart contracts written in Prolog. The primary focus is to overcome the need for smart contracts on existing blockchain systems such as Ethereum to be manually invoked; this leads to the "proactive" contract. A secondary focus is on the use of the logic programming paradigm, which we analyse here.

The smart contract system of Maffi's work can essentially be seen as a decentralised logic interpreter. Interaction with smart contracts takes the form of blockchain transactions containing a goal, in the same way one submits goals to a Prolog interpreter. Contract code is also made mutable through Prolog's meta-programming features, which are advocated as a way for developers to fix bugs and push updates.

```
1  % Facts which store the necessary identities.
2  acme_id(o38bn27y4).
3  cervice_id(d92jf4dfs).
4  acme_smart_device_id(g21vfjd4c).
5  globox_smart_device_id(y5dv7pc21).
6  shipment_cost(50).
7
8  receive(departure(UUID)) :-
9     sender(Sender),
10    acme_smart_device_id(Sender),
11    shipment_cost(Amount),
12    assert(to_pay(UUID, Amount)), % remember to pay
13    delayed_task(3 * 86400, partial_refund(UUID)), % 3-day delay
14    delayed_task(5 * 86400, total_refund(UUID)). % 5-day delay
15
16 receive(arrival(UUID)) :-
17    sender(Sender),
18    globox_smart_device_id(Sender),
19    retract(to_pay(UUID, Amount)), % if this rule is still present...
20    cervice_id(Id), % ...then remove it...
21    transfer(Id, Amount). % ...and pay Cervice
22
23 partial_refund(UUID) :-
24    retract(to_pay(UUID, Amount)), % if this rule is still present...
25    Refund is Amount * 0.40,
26    acme_id(Id),
27    transfer(Id, Refund), % ...refund 40% to ACME...
28    Remaining is Amount * 0.60,
29    assert(to_pay(UUID, Remaining)). % ...and store the remaining
30
31 total_refund(UUID) :-
32    retract(to_pay(UUID, Remaining)), % if this rule...
33    acme_id(Id), % ...is still present, remove it and...
```

```
34     transfer(Id, Remaining). % ...refund the full amount to ACME
```

Listing 3.2: Smart Contract For Supply Chain Management [40]

Listing 3.2 shows an example of a smart contract taken from the paper. This scenario concerns the shipment of products from a manufacturing company ACME to a company Globox. When a product is dispatched from ACME, a smart device reads its RFID tag and automatically sends a transaction to the contract with the goal `departure(UUID)`, where UUID is a unique identifier for the product. When it then arrives at Globox, a similar smart device sends a transaction with the goal `arrival(UUID)`. Depending on how long shipping took, ACME may receive a partial or total refund of the shipping cost.

In Maffi's design, when a transaction with some goal `Goal` is sent to a contract, the rule of the form `receive(Goal) :- Body` with the matching `Goal` is triggered, and the computation of `Body` is spawned. For `departure(UUID)`, the contract checks that the transaction sender is the ACME smart device, and asserts a fact into the program initially indicating that the shipping cost is to be paid in full to Cervice. Two computations are then scheduled: if the shipment is not delivered in three days time, a partial refund is arranged, and if the shipment is not delivered in five days, a total refund. On `arrival(UUID)`, it checks that the sender is the Globox smart device, retracts the fact indicating the amount to be paid to Cervice, and transfers funds accordingly. (Note that this is just a proof of concept – it is not actually possible to transfer money with his implementation.)

Maffi suggests that the choice of a declarative language may improve their comprehension and ease development. However, we contest that his contracts are not really written in a declarative style. In the above contract for example, the rule bodies depend on side-effecting predicates like `assert` and `retract`. The result is that the clauses of the program only have a procedural reading: "when invoking `partial_refund(UUID)`, do retract X, transfer Y, assert Z". They lack the declarative reading of a logic statement: "partial refund for UUID is the case if ...". The benefits of the logic-based formalism are lost, since the program's clauses no longer constitute a high-level specification; they instead devolve into a series instructions resembling the statements of an imperative language.

Implementation-wise, the system relies on Tendermint [15] to handle blockchain management and peer-to-peer connectivity. Each node in the network runs a logic interpreter written in Java that interfaces with a tuProlog logic engine. The blockchain application state σ consists of the collection of all smart contracts. A transaction takes one of four types: create, invoke, update or delete, which create a new contract, submit a goal to a contract to trigger computation, update the source code of a contract and remove a contract respectively.

Evidently, a limitation is the lack of a cryptocurrency. This severely restricts the power of the system's smart contracts, as they are not able to hold, send or receive money. Another concern is the issue of trust that arises from enabling contracts to be updated and deleted. Immutability of smart contracts on platforms like Ethereum is often celebrated as one of their primary advantages. Making the code mutable invites the possibility for a malicious party to alter the terms of the contract.

The use of Prolog in a blockchain environment raises a number of issues which Maffi has been perspicacious to identify. One is the possibility of infinite computation – a Prolog query may not terminate. Maffi addresses this by enforcing an execution time limit: before a transaction is included in a block, a simulated execution of the operation is run. If this exceeds the time limit, the transaction is simply discarded. Another issue is of non-determinism – certain Prolog predicates such as those for random number generation can give unpredictable outputs. This is solved by removing these library predicates from the Prolog engine.

To our knowledge, Maffi has been the first and only to provide a complete implementation of a blockchain platform that supports the execution of general, logic-based smart contracts. He highlighted that Prolog's backward reasoning enables blockchain transactions to be modelled as logic goals. Nevertheless, we have seen there is much to be desired. An interesting smart contract language would preferably detach the rules of the contract from the state-modifying actions, so as to retain the benefits of a declarative formalism. The supporting blockchain platform would ideally preserve immutability of smart contract code and include a native currency which contracts can operate.

3.3 Towards Declarative Smart Contracts (2020)

Purnell and Schwitter [41] use an Answer Set Programming (ASP) representation of smart contracts to address the difficulties in translating legal documents. The idea is that ASP programs are simpler and more amenable to automatic code generation than their Solidity counterparts. This has allowed the authors to build an interactive user interface for the automatic translation of legal documents to smart contract code.

Answer Set Programming is advocated because its support for weak and strong negation allows it to model legal reasoning intuitively. An ASP program is split into three components: the facts, the logic program and the events, a structure which resembles legal documents.

In Purnell and Schwitter's design, smart contract execution is triggered by events, which are ASP programs containing a single fact. The fact is indicative of some external occurrence at some point in time. For example, one might send an event containing the program `death("fred wallace", 43677)` to a smart contract representing a will and testament to signify a death. Upon receipt, the system combines the facts, logic program and events that make up the contract, and executes an ASP solver. This yields an answer set containing atoms which represent the actions to performed. For example, the predicate `transfer` represents a monetary transfer. The authors envisage an interface for translating the answer sets into commands executed by Ethereum.

Answer Set Programs support many extensions to the standard logic programs of Horn clauses, such as integrity constraints, strong negation, choice rules, cardinality constraints and optimisation statements. These features are useful for domain knowledge representation in constraint solving and optimisation problems. However, it is not clear that they are particularly useful for modelling contracts. It is uncommon for contractual terms to include preferences that would warrant the need for optimisation statements, for example. Indeed, Purnell and Schwitter appear to assume that their smart contracts only admit a single answer set, which suggests the features of disjunctive heads go unused and that the contracts could perhaps be expressed as stratified logic programs instead. In this case, the contracts could be more efficiently computed by stratified extensions to Datalog or Prolog.

Purnell and Schwitter present an implementation of their software and user interface for translating legal documents to ASP contract code. However, no blockchain system is implemented to support their execution. The authors allude to the embedding of an ASP solver within the Ethereum Virtual Machine, but the feasibility of such an approach is unclear.

3.4 A Method of Logic-based Smart Contracts for Blockchain Systems (2018)

Hu and Zhong [42] present Logic-SC, a smart contract language based on Active-U-Datalog [43] with temporal extensions. They criticise the use of defeasible logic (such as in [5]) due to its inability to express temporal factors and the lack of a triggering mechanism for the execution of contract procedures. By introducing temporal active rules, Active-U-Datalog overcomes these limitations.

Figure 3.1 shows an example contract taken from the paper. A Logic-SC contract consists of a set of constants *CONST*, an extensional database *EDB* (empty in this case), an intentional database *IDB* and a set of temporal active rules *TAR*. *EDB* and *IDB* are as in Active-U-Datalog [43]. Together, they essentially form a logic program but with one extension: the intensional rules can have update atoms in the body. To briefly explain the temporal active rule by example, consider the last rule, rule (10). On every change of date, the system inserts a predicate `date(d)` into the program, with `d` being the current date. Because rule (10) contains the special update atom `+date (d)` in its body, this rule is triggered. If the other literals in the body hold at this point, the head predicate `unfinished_tag(A,B)` is inserted into the EDB. Thus this rule says: if at the completion date a certain `check` fails to pass, B pays A 3000 dollars and a tag is inserted into the database indicating an unfinished status.

```
(1)    Contract (
(2)    ProjA,
(3)    {  signed_date≡' 10/01/17'; completion_date≡'12/31/17';
(4)       subscription-day≡([10/01/17, 12/31/17], {1}Days ▷ 7.Days); },
          //CONST
(5)    {   }, //EDB
(6)    {  check(A, B)←input_result(A, r), r='qualified'; }, //IDB
(7)    {   +initial_ payment_tag(A, B)←+date(d), d∈deposit-day, ¬initia
           l_ payment_tag(), pay(A, B, 2000);
(8)        +cancel_tag(A, B)←+date(d), d>signed_date, d∉ deposit-day,
           ¬initial_ payment_tag(), cancel(A, B);
(9)        +finished_tag(A, B)←+date(d), d=completion_date, check(A,
           B), pay(A, B, 3000);
(10)       +unfinished_tag(A, B)←+date(d), d=completion_date, ¬check
           (A, B), pay(B, A, 3000); },  //TAR
(11)   )
```

Figure 3.1: Logic-SC Smart Contract [42]

Logic-SC contracts are rather flexible. Execution can be triggered both by external transactions (which trigger the intensional rule with the matching head), and automatically at certain times (by special body predicates in temporal active rules). Actions also take multiple forms. Database updates are written as update atoms which can appear both in the body of *IDB* rules and the head of *TAR* rules. However, it appears that non-update atoms can generally also create side effects, as is the case for the predicate `pay/3`.

This flexibility perhaps comes at the price of readability and style. Let us consider again rule (10). Although written to resemble a logical implication, the rule does not have a clear "trigger-condition-consequence" structure. The two actions `+unfinished_tag(A, B)` and `pay(B, A, 3000)` are separated at opposite ends of the clause. One is in the head and the other is in the body, despite both intuitively being "consequences". Additionally, while the plus-prefixed atom `+date(d)` in this context represents the "trigger", the same atom written in an *IDB* rule would have the complete opposite meaning of an action to add `date(d)` to the database. The rules regarding the placement of the triggers, conditions and actions are inconsistent, which creates confusion for both the development and comprehension of Logic-SC smart contracts.

The concept of temporal triggers is also problematic in general. Hu and Zhong require the system to trigger an hourly time tick on every smart contract. Each tick spawns a potentially non-trivial computation. As the number of contracts grows, this quickly demands a significant amount of the blockchain's computational resources. In the worst case, the network is too busy processing these ticks and becomes unable to service new transactions, resulting in a denial of service. The paper only gives scant implementation ideas and does not recognise this issue.

Active-U-Datalog was not designed to represent smart contracts – it is a database update language [43]. As such, it may not be the best choice for representing contracts. The aforementioned oddities arise because Hu and Zhong's addition of temporal triggers and external actions for monetary transfer especially do not integrate well with its syntax of update atoms. We still, however, desire the ability to express facts and logical implications in the way that Active-U-Datalog and Logic-SC enable with their extensional and intensional databases. A more suitable smart contract language might keep the *EDB* and *IDB* but replace the temporal active rule with a more domain-specific construct.

3.5 Logic-based Smart Contracts (2020)

Stancu and Dragan [44] discuss the advantages of the logic programming approach to smart contracts. They proceed to outline a blockchain implementation supporting smart contracts written

in Prolog for controlling the creation and management of application licences.

The use of Prolog is advocated to mitigate security vulnerabilities. It is deemed closer to the natural language used to describe business logic than its procedural counterparts, simplifying the translation of contractual clauses to code. However, the authors' commentary is limited to these general advantages, which they do not demonstrate. Discussion on the smart contracts of their own writing is minimal, and the Prolog contracts of their implementation are absent from the paper. It can be gathered from their description that the smart contracts are merely used to determine when a licence can or cannot be transferred; it is not possible to define more complex interactions (e.g., determining when a party has breached the licence, etc).

The authors instead place more focus on the architecture of their implementation. Each node of their blockchain runs a server containing an instance of BigchainDB [45] for the storage of user accounts and licences. Tendermint [15] ensures the consistent view of the BigchainDB application across the network. When a smart contract is invoked, the server runs an instance of SWI-Prolog, consults the program and evaluates its clauses. Python is used to interface between the database and SWI-Prolog.

Stancu and Dragan's system only supports the specific use case of application licences. Prolog is used to verify whether a licence can be transferred against a limited set of predefined conditions. Although the authors suggest the system can be extended to other use cases, it does not appear to be possible to enforce generic agreements written in Prolog. To enable users to use new clauses to express more complex contract conditions requires the explicit support of those clauses by manual addition on the part of the system developers.

3.6 Others

This section is dedicated to other work in the area of logic-based smart contract languages which do not add much to the discussion of this chapter, either because they are only tangentially related, or because information is scarce.

Logikon [46]: Logikon is described as a logical-functional language inspired by Prolog and Haskell. It compiles to Yul [47], which in turn compiles down to EVM bytecode. There have been no updates since 2018. No documentation for the language is provided beyond a handful of short examples, one of which is displayed in listing 3.3.

```
declare Balance Array.
declare Owner Uint.

define isOwner (Addr) _ :- (= Addr Owner).

define transfer (Uint Uint) -> Bool
case (To Amount) _ :-
    (= FromBalance (select Balance Sender))
    (>= FromBalance Amount)
    (= ToBalance (select Balance To))
    (= Balance2 (store Balance Sender (- FromBalance Amount)))
    (= Balance3 (store Balance2 To (+ ToBalance Amount)))
    (prove (= (sum Balance3) (sum Balance)))
    (update Balance Balance3).
```

Listing 3.3: Logikon Transfer Example [46]

LPS [48]: Logic Production Systems (LPS) combines the reactive rules of production systems with the clauses of logic programming, giving a logical interpretation to an imperative language. In LPS, execution has the objective of model generation: the reactive rules define goals, and the system performs actions to generate a model in which the goals are satisfied. Although not originally designed to write smart contracts, its usage for this purpose has been explored. See listing 3.4 for an example of a bank transfer. LPS has not been investigated further in the interest of time, but it presents a potentially interesting alternative.

```
maxTime(10).
actions   transfer(From, To, Amount).
```

```
3 fluents   balance(Person, Amount).
4
5 initially balance(bob, 0), balance(fariba, 100).
6 observe   transfer(fariba, bob, 10)   from 1 to 2.
7
8 if      transfer(fariba, bob, X)  from  T1 to T2,
9         balance(bob, A) at T2, A >= 10
10 then    transfer(bob, fariba, 10)   from T2 to T3.
11
12 if      transfer(bob, fariba, X)  from  T1 to T2,
13         balance(fariba, A) at T2, A >= 20
14 then    transfer(fariba, bob, 20)   from  T2 to T3.
15
16 transfer(F,T,A) updates Old to New in balance(T, Old) if New is Old + A.
17 transfer(F,T,A) updates Old to New in balance(F, Old) if New is Old - A.
18
19 false transfer(From, To, Amount), balance(From, Old),  Old < Amount.
20 false transfer(From, To1, Amount1), transfer(From, To2, Amount2),  To1 \=To2.
21 false transfer(From1, To, Amount1), transfer(From2, To, Amount2),  From1 \= From2.
```

Listing 3.4: LPS Bank Transfer Example [49]

L4 [50]: Aimed at being the "SQL for contracts", L4 is smart contract programming language for Ethereum based on deontic modal logic. Almost no information could be found about the language itself.

3.7 Summary

As we have seen, the logic-based languages of the related work are significantly varied. Hu and Zhong's Logic-SC is based on extensions to Datalog. Maffi and Stancu and Dragan use off-the-shelf Prolog. Purnell and Schwitter consider Answer Set Programming. Governatori et al. and L4 propose deontic logic. Logikon fuses functional programming and logic programming. Finally, LPS combines production rules with logic programs.

Implementation options, however, have not been as comprehensive. Governatori et al. compared off-chain and on-chain inference solutions. Purnell and Schwitter focus on the user interface rather than contract execution. Maffi and Stancu and Dragan use Tendermint to create their own blockchains. Others lack a working implementation.

Each related work provides a unique insight into the arguably underexplored field of logic-based smart contracts. However, each also comes with its set of issues and challenges. Amidst the ideas presented is the scope for the formulation of a new language, which can simultaneously address these pitfalls and deliver an implementation to demonstrate its feasibility. The design of this language is the subject of the next chapter, Chainlog.

Chapter 4

Chainlog

This chapter introduces Chainlog, the main contribution of this work. The name "Chainlog" is a portmanteau of "blockchain" and "logic", and is used to describe both the language and its supporting blockchain. We refer to the language simply as Chainlog, and the blockchain as the Chainlog Platform.

We start by determining the requirements of both the language and the blockchain platform. Then, we give a definition of the language's syntax and semantics. Finally, we provide a specification of the blockchain's important data structures and state machine.

4.1 Requirements

4.1.1 Language Requirements

There exist many classes of logic programs. This section seeks to define and justify the required features and computation model of the logic language. We assume as base the most elementary kind of logic program, that of propositional definite clauses, and motivate the extensions necessary for the representation of contracts through the use of examples.

We begin by remarking that propositional logic is not sufficient. Consider a very simple example of a smart contract for warehouse management, which tracks when a company's inventory runs empty across a range of raw materials. In propositional logic, these would take the form of facts $steel_empty$, $aluminum_empty$, $rubber_empty$, etc. Suppose we would like to know whether any of our stock is empty so that we can order more. Within the confinements of propositional logic, we would need to check the truth value of each propositional fact one by one. To alleviate this, we require some features of *first-order logic*: notably, the ability to quantify over variables. We would like to be able express these facts as predicates $empty(steel)$, $empty(aluminum)$, $empty(rubber)$ so that we can ask "is there an X such that $empty(X)$?", or $\exists X(empty(X))$.

The introduction of variables implies the need for *unification*. Suppose our warehouse smart contract is currently the logic program $\{empty(steel)\}$. When we ask of the program the query $\exists X(empty(X))$, we expect the answer to be true, even though the predicates $P = empty(steel)$ and $Q = empty(X)$ are not structurally identical, because there exists the substitution $\theta = \{X/steel\}$ for which $P\theta = Q\theta$. As a user, we would like to know not just that the query is true, but also for which X the query succeeded – so that we know which materials to order. Hence, the logic engine requires a unification algorithm to (1) identify that substitutions exist unifying two predicates and (2) report the substitutions to the user.

First-order definite clauses do well to capture the rules of a contract. Take, for example, a travel insurance policy. A smart contract used to determine whether or not a policyholder qualifies for a

payout may do so through clauses like the following:

$$qualifies(baggage_cover, Policyholder) \leftarrow baggage_lost(Policyholder)$$
$$qualifies(medical_cover, Policyholder) \leftarrow hospitalised(Policyholder)$$
$$\vdots$$

It is common for a contract to have some notion of exceptions to its rules. The insurance policy might further specify, for example, that the medical cover does not apply if the policyholder was engaged in a high-risk activity such as skydiving, bungee jumping, paragliding or rock climbing. This is the classic qualification problem; it necessitates negation. Rather than support classical negation, we shall instead require *negation as failure (NAF)* as it can be implemented efficiently in the logic programming framework [51]. The insurance contract's medical cover can then be expressed in a manner such as the following:

$$high_risk(skydiving)$$
$$high_risk(bungee_jumping)$$
$$high_risk(paragliding)$$
$$high_risk(rock_climbing)$$

$$\begin{aligned} &qualifies(medical_cover, Policyholder) \leftarrow \\ &\quad hospitalised(Policyholder), \\ &\quad activity(Policyholder, Activity), \\ &\quad \text{not } high_risk(Activity) \end{aligned}$$

Notice the use of "not" in the last body condition, which succeeds when there is no evidence to suggest that *Activity* is high-risk. This kind of reasoning is appropriate because we can make the closed-world assumption: if the program does not say an activity is high-risk, we assume it is not high-risk.

In terms of computation of logic programs, there exist two broad methods: forward and backward reasoning. In the forward or bottom-up approach, computation starts at the program's facts and generates new consequences by repeatedly applying the rules. On the other hand, the backward or top-down approach starts with a goal to be solved and attempts to prove it by repeatedly searching for rules in the program that can justify it. The backward approach is more suited to the blockchain query and transaction model. There is a natural parallel between blockchain queries and logic queries, which is similar to how Maffi [40] modelled the interaction with smart contracts as the submission of logic goals. When the goal to be solved is known, bottom-up inference typically derives a lot of irrelevant facts, whereas in the top-down approach the goal restricts the search space. For this reason, we opt for the use of backward reasoning.

In principle, a smart contract should be receptive to new information over time. Thus, the language must possess features that allow the program's beliefs to be updated. To use the travel insurance example above, in the event that the policyholder does become hospitalised, the smart contract needs some way of being informed that $hospitalised(Policyholder)$ becomes true after the contract is deployed. To achieve this, it suffices to require the ability to insert and remove facts from the logic program. (To permit the addition and deletion of the program's rules would imply the ability to tamper with the contractual terms.)

The language must also include constructs to support the receipt and transfer of money. Imagine a decentralised gambling smart contract where users bet on a dice roll. Users need to be able to send their bet to the smart contract. In addition, the language needs to be able to express the action of transferring winnings to the lucky user.

Finally, we desire a way to distinguish the actions from the purely declarative clauses of the program. In the same way that Solidity's `view` and `pure` keywords identify the procedures which do not modify state, we would like some way of determining whether an execution will be free of side-effects. This would enable the pure predicates of the program to be callable via blockchain queries. Users would be able to leverage the power of logic queries without the need for a transaction.

Let us now consider some of the logic programming languages used in practice.

Datalog is one of the simpler and more restricted logic languages in use. Pure Datalog consists of definite clauses without function symbols. It supports variables and unification, and can be evaluated both top-down and bottom-up. While pure Datalog lacks features for updating the database and transferring money, Hu and Zhong's Logic-SC [42] overcomes this by allowing update atoms in the head and body of certain rules, and by allowing predicates to spawn side effects. This was argued to be confusing as it caused the conditions and actions to be intermingled. An appealing property of Datalog is that execution is guaranteed to terminate; however, this means that negation as failure is not supported.

Prolog does support negation as failure. It also uses a backward chaining execution model. Both Maffi [40] and Stancu and Dragan [44] have applied Prolog for smart contract applications with some success. Although Prolog computations are in theory not guaranteed to terminate, practical limits can be imposed either through a gas model like Ethereum or a simple time limit like in Maffi's system. The problem with Prolog is that in order to be "useful", programs often become littered with impure predicates which sacrifice their declarative nature. We saw this with Maffi's use of side-effecting predicates like `assert` and `transfer`, but this is observed more generally by the use of Prolog's more contentious constructs for controlling search, such as the infamous cut (`!`), if-then-else (`-> ;`) and exception handling features (`catch, throw`). A truly logic-based language would exclude such features.

Answer Set Programming is utilised by Purnell and Schwitter [41] in the context of legal contracts. We argued that many of ASP's extensions such as strong negation and choice rules are generally not useful for modelling contracts. Their inclusion would entail the need to reason about extended and/or disjunctive logic programs, which would unnecessarily complicate the semantics and slow down execution. ASP programs are also executed bottom-up through a process of grounding and solving, meaning they suffer from the drawback of forward reasoning described earlier.

Governatori et al. [5, 39] advocated the use of the deontic defeasible language FCL due to its applicability in modelling legal reasoning. We remarked that this was too constrained, and desired a more general logic framework in order to express a wider range of contracts. It turns out that the metalogical capabilities of standard logic programming languages are powerful enough to express deontic notions [52, 53], without being restricted to this highly specialised logic. We therefore choose to dismiss a deontic formalism.

It is apparent that neither the existing logic languages nor the related work on logic-based smart contracts fully satisfy our requirements and objectives. A new language is in order.

4.1.2 Blockchain Requirements

Governatori et al. [5, 39] touched on some of the challenges of integrating a logic language into existing blockchain platforms. Smart contract execution is completely isolated, so it is not possible to simply call to a logic interpreter. This led to the suggestion of clumsy solutions such as the movement of computation to off-chain inference engines or the encoding of a metaprogram in existing smart contract languages.

An alternative approach taken by Maffi [40] is to build a new, bespoke blockchain that is specifically designed to execute logic programs. This is the approach that is taken in this work. The remainder of this subsection seeks to outline the requirements for such a blockchain system.

At minimum, the system must meet the functional requirements of a blockchain: it must consist of a network of nodes which cooperate to maintain a replicated state machine. Each node must store a local copy of the application state. A consensus strategy is required to coordinate this state. Nodes must listen for and respond to queries. They must also listen for, execute and propagate transactions. A record of all transactions must be kept by grouping them into blocks connected in an ordered blockchain data structure.

To serve as a logic-based smart contract platform, the blockchain must have facilities to (1) deploy new smart contracts; (2) expose the source code of existing smart contracts, a requirement for transparency; (3) submit queries to existing smart contracts in order to retrieve information about their state and (4) trigger execution of contract code so as to enforce contractual terms. In addition,

the blockchain should enforce a number of important properties of smart contracts. The contract terms should be immutable, to prevent tampering. All execution of smart contract code must be deterministic, so that consensus can be reached among the network's nodes. Finally, execution must always terminate, in order to prevent denial of service attacks.

The blockchain platform must include a scheme to identify its participants – end users and smart contracts alike need to be able to refer to each other in a non-ambiguous manner. Specifically, each end-user and smart contract is required to be endowed with a unique address from some set of valid addresses *Addr* (which shall be defined later). The platform must also include a scheme for authentication – there must be a way to verify that a participant is who they claim to be.

Many contracts are concerned with money. Therefore, the blockchain shall be required to have a native cryptocurrency. The platform shall expose capabilities for transferring money, both between end-users and also to and from smart contracts.

4.2 Language Design

One of the primary challenges with logic-based languages is that while they are great for knowledge representation, they are not well suited to expressing the actions of real-life contracts. Attempts to bridge this gap in existing logic frameworks have been the source of inadequacies, often leading to the forfeiture of the declarative expression, as we have seen in related work.

Chainlog is a hybrid logic-based and imperative language. It is built on the idea that the logical knowledge base and the procedural commands should be clearly separated. By instituting this demarcation, the knowledge base remains purely declarative, and therefore enjoys the associated benefits: it reads as a high-level specification, is easier to reason about, mitigates programming errors and is queryable in its entirety from outside the blockchain. Yet, the power to express the actions and effects is not lost – it is endowed by the imperative component.

4.2.1 Syntax

A Chainlog program P is a triple $\langle KB_s, KB_d, MH \rangle$, where:

- KB_s is the *static knowledge base.*
- KB_d is the *dynamic knowledge base.*
- MH is the set of *message handlers.*

In theory, the static knowledge base KB_s is a normal logic program. That is, a set of clauses of the form:

$$A \leftarrow B_1, ..., B_m, \text{ not } C_1, ..., \text{ not } C_n \qquad (m \geq 0, n \geq 0)$$

where A, B_i and C_i are all atoms and "not" denotes negation as failure. In practice however, the clauses of KB_s are ordered, and are written in the Prolog form `A :- B, not C.` where atom, predicate symbol and functor names begin with a lowercase letter and variable names begin with an uppercase letter or underscore.

The dynamic knowledge base KB_d is a set of atomic facts. These may be non-ground, in which case variables will have implicit universal quantification. In practice, the facts are ordered.

MH is an ordered set of zero or more message handlers. A message handler is an imperative construct with a syntax loosely inspired by event-condition-action (ECA) rules:

`on` $handlerTerm$
    (`if` $condition$ | `require` $condition$)$*$
    `do` $action$

The handler term $handlerTerm$ is a logic term belonging to an implicitly defined signature consisting of constants, variables and function symbols. In other words, it can be a constant (e.g. `donate`, `claimPayout`), a variable (e.g. `X`, `_`) or compound of terms (e.g. `vote(Candidate)`, `register(name (First, Last), male, Age)`). Variables have implicit universal quantification. As a convention, the atom or functor name will be written in camel case throughout this report.

The body of the message handler contains of zero or more `if` or `require` clauses. (Note: this use of the word "clause" is not to be confused with the logic sense of the word; these clauses are not disjunctions of literals.) In both cases, the *condition* is a goal, which may be a single literal or conjunction of comma separated literals. Variables have implicit existential quantification. The difference between the two will be explained in the semantics section.

Finally, the body must end with exactly one `do` clause. The *action* may be one of the following:

- `assert`$(Term)$, where $Term$ is a constant, variable or compound term.
- `retract`$(Term)$, where $Term$ is as above.
- `transfer`$(ToAddress, Value)$, with $ToAddress \in Addr, Value \in \mathbb{N}$.
- $action_1,\ action_2$, a composite of actions where $action_1$ is one of the above three actions and $action_2$ is one of the above three actions or another composite. The comma (,) operator in this context is therefore right-associative.

We shall denote *Actions* to be the set of actions defined by the above rules.

We conclude this subsection by introducing some helpful terminology.

- $KB_s \cup MH$ is the *source code*. This is what the programmer writes and deploys. It is the immutable part of the program. (KB_d is the part that is updated over time.)
- $KB = KB_s \cup KB_d$ is the *logic program* or *knowledge base*. This is the declarative component of the program. (MH is the imperative component.)

4.2.2 Semantics

The semantics of a Chainlog program is naturally divided into the semantics of the logic program and the message handlers.

The declarative semantics of normal logic programs has been well studied in e.g. the stable model semantics, so it is not belaboured here. We will also see that the search strategy employed by Chainlog is necessarily incomplete with respect to these semantics.

The meaning of a Chainlog program is best understood by its operational semantics. Fundamentally, there are two ways an end user interacts with a Chainlog smart contract: by blockchain queries and blockchain transactions. Blockchain queries can be thought of as abstractions around logic queries to the program's knowledge base; the operational semantics of the logic component describes these queries. Blockchain transactions on the other hand are used to trigger a contract's message handlers; the operational semantics of the message handlers describes how these are processed.

<u>Operational Semantics of the Logic Program</u>

The logic program's operational semantics is a modification of Prolog's implementation of the SLDNF resolution procedure [54], which will be referred to as SLDNF*.

Like SLDNF, SLDNF* starts with the negated goal and attempts to reduce it to the empty clause. SLDNF* follows Prolog's depth-first strategy for reasons of space efficiency. Goal literal selection is left-to-right; program clause selection is top-to-bottom. A negative literal in Chainlog is safe if it is ground on selection. In addition, SLDNF* omits the occurs check due to its computational expense.

The primary difference between SLDNF and SLDNF* lies in the need to avoid non-terminating computation. Deep consideration has been given towards the method most suitable for evading non-termination as part of this work. The halting problem has shown that determining whether a program terminates is undecidable, so it is not possible to analyse this upfront. Alternative search strategies like breadth-first search and iterative deepening can avoid getting stuck in infinite branches in some cases, but they are less efficient and the inclusion of functions means they can still fail to terminate when there are no solutions to the query. One could instead consider an artificial restriction such as an execution time limit (as in Maffi's solution [40]) or a pricing/gas model (as in Ethereum [14]). The problem with a time limit is that it depends on hardware, which means it is possible that one blockchain node reaches the limit but another finds a solution for the same query – a problem for establishing consensus.

Constructing a full-fledged gas model would adequately solve this problem, but is beyond the scope of this work. Instead, SLDNF* currently imposes a depth limit of 4096 on the proof tree. Each resolution with a program clause increases the depth by one. Unlike execution time, a limit on depth is deterministic in whether it is exceeded or not for a given a program, query and otherwise deterministic search strategy. When the depth limit is reached and a further inference is attempted, SLDNF* will immediately backtrack to the topmost query and report it to be "unknown" (since in principle it cannot be determined whether the query is true or false). Practically, this is an exception. Note that this is not the same as the depth limited search strategy, which upon reaching the depth limit may still backtrack and explore alternative branches.

During clause selection, SLDNF* looks in four places: the static and dynamic knowledge bases of the Chainlog contract KB_s and KB_d, a knowledge base of built-in/library predicates KB_l and what is called the context knowledge base KB_c. These are consulted in a strict order $KB_l \prec KB_s \prec KB_c \prec KB_d$. Let us use this opportunity to explain some of the special and domain-specific predicates in KB_l and KB_c a Chainlog contract has access to.

The library KB_l includes predicates that are useful for contracts in general, and/or cannot be implemented in pure Chainlog. Notably, this includes:

- **Mathematical predicates** such as `is`, `>`, `>=`, `<`, `=<`, etc. `is` is as in Prolog and interprets arithmetic operators `+`, `-`, `*`, `/`, etc. as one would expect.
- **Unification predicates** `=` and `\=`.
- **Between**, a predicate `between(+Lower, +Upper, ?Value)` which can be used to check `Lower` $<$ `Value` $<$ `Upper` or to generate integers in the given range.
- **List processing utilities** such as `append`, `length`, `member`, but also `sum_list`, `max_list`, `min_list`.
- **Time processing utilities**, which includes facts representing various durations of `seconds`, `minutes`, `hours`, ..., predicates for date arithmetic, etc.
- **The second order predicate** `find_all`, equivalent to Prolog's `findall`.
- **Aggregation utilities** `count_all`, `sum_all`, `max_all`, `min_all`. Note that these are just for convenience and could be defined in pure Chainlog: `count_all` is just `find_all` plus `length`, `sum_all` is just `find_all` plus `sum_list`, and so on.
- **Disjunction** via the predicate `or`, which may be written infix. Aside from its operator status, this is again a convenience that could be defined in pure Chainlog:

```
or(P, _) :- P.
or(_, P) :- P.
```

Unlike the library, the context knowledge base KB_c is transient: it is constructed just before execution and destroyed immediately after. Its purpose is to inject information about the current state of the blockchain. For a blockchain query, this only contains two unary facts: `time` contains the (approximate) current UNIX timestamp and `balance` contains the contract's cryptocurrency balance.

It must be highlighted that Chainlog is not a subset or superset of Prolog, despite sharing similar syntax, semantics and some library predicates. The same is true with regards to Datalog. Chainlog

is a domain-specific language for representing smart contracts. Care has therefore been taken to restrict the logic component of the language and exclude many of the irrelevant or harmful features present in Datalog and Prolog, for example:

- **Search controlling predicates** such as Prolog's cut (`!`) and if-then-else (`-> ;`). We desire the logic program to be purely declarative and these predicates would undermine this property.
- **Exception handling** like Prolog's `catch` and `throw`, for similar reasons.
- **Database update predicates** such as update atoms in Datalog extensions or the `assertz`, `retract` and `abolish` family of predicates in Prolog. Chainlog leaves database updates to the message handlers in order to keep the logic program side-effect free.
- **Program loading predicates** like Prolog's `consult`. Smart contracts should execute in isolation and have no access to the filesystem.
- **I/O predicates** including Prolog's `open`, `write`, `read`, `close` and similar predicates. Again, to maintain isolation.
- **Exit predicates** such as `halt`. The knowledge base should not be concerned with when to exit the process.

Operational Semantics of Message Handlers

End-users invoke a smart contract by sending it a message, which we will see later is essentially a wrapper around a ground term $messageTerm$ and a cryptocurrency payload. The message is processed by the message interpreter in two phases. In the first phase, $messageTerm$ is resolved with the program's set of message handlers MH, which yields an $action \in Actions$ to be performed if successful, or an exception otherwise. If the first phase is successful, the second phase effects the *action*, propagating the necessary changes to state.

For the duration of a message, an ephemeral context knowledge base KB_c is populated with four unary facts: `sender`, `value`, `time` and `balance`. These endow the smart contract with knowledge of who the message sender is, how much money was attached, the (approximate) current UNIX timestamp and the smart contract's current balance.

Phase 1

The first stage works as follows. The interpreter searches the program's message handler set MH top-to-bottom and attempts to find a handler whose $handlerTerm$ unifies with the $messageTerm$. When such a handler is found, the most general unifier $\theta = mgu(messageTerm, handlerTerm)$ becomes the "current unifier", and the interpreter then processes the body clauses in sequence. The unifier is accumulated through this execution.

When encountering an `if` *condition* or `require` *condition*, the current unifier is applied to the *condition* and it is submitted as a query to the SLDNF* procedure. If SLDNF* returns success, the clause passes. The resulting unification is composed with the current unifier and propagated; execution proceeds to the next body clause.

In the successful case, execution reaches all the way down to the final `do` *action* clause with some current unifier θ composed of all the unifications made up to this point. The first phase concludes with $action\theta$ (the unifier applied to the action) as its result.

The difference between `if` and `require` concerns their behaviour when the respective *condition* is false: `if` is a "soft" failure, while `require` is a "hard" failure.

- When an `if` condition fails, execution has the opportunity to backtrack, e.g. to a previous `if`. This is essentially the same behaviour as a Prolog body clause.
- `require` is analogous to the Solidity statement of the same name. When a `require` condition fails, a `require_error` exception is raised. The phase terminates immediately with this exception.

The purpose of `require` is to validate *user-facing* conditions. For example, you would use `require` to check that a user sent enough money to the contract, or that the user qualifies for a payout when making an insurance claim. The `require_error` thrown will include the condition that failed; the idea is that this will get returned to the user so that they can see why the message was unsuccessful.

An `if` should be used in other cases, when failure of the condition should not be reported to the user. For example, to retrieve a fact in the program that you know to be true, or to perform a mathematical calculation. `if` is also appropriate if backtracking is desired.

Note that if the SLDNF* query of the condition returns the "unknown" exception, i.e. if SLDNF* reached the depth limit, both the `if` and `require` constructs will propagate this error and fail the first phase with this exception.

A message handler is said to match for a given $messageTerm$ if the $handlerTerm$ unifies with the $messageTerm$ and all `if` and `require` conditions succeed. As the message handlers are searched top-to-bottom, in the case that multiple message handlers match, the actions of only the topmost handler will be returned. If none of the program's message handlers match, the first phase ends with the `match_error` exception.

Some examples follow to illustrate the procedure.

Example 1

$candidate(adam)$
$candidate(emily)$
$candidate(sophie)$

`on` $castVote(Cand)$
    `require` $candidate(Cand)$
    `do assert`$(voted_for(Cand))$

Shown above is the source code, i.e. the $KB_s \cup MH$, where KB_s consists of the three $candidate$ facts and MH consists of the single message handler. Assume KB_d is empty.

When a message with term $castVote(adam)$ is received:

- MH is searched. In this case there is only one choice. The handler term $castVote(Cand)$ unifies with the message term $castVote(adam)$ with $\theta = \{Cand/adam\}$. Hence, θ becomes the current unifier, and execution proceeds to the handler's body.

- `require` $candidate(Cand)$ is encountered. The current unifier θ is applied to the condition to give $candidate(adam)$. This is then submitted to SLDNF*, which finds this to be true in KB_s and returns success. The query made no new unifications, so execution proceeds with the current unifier θ.

- `do assert`$(voted_for(Cand))$ is reached. The unifier θ is applied to the action. The phase concludes successfully, returning the action `assert`$(voted_for(adam))$.

Now suppose instead a message with term $castVote(liam)$ is received:

- MH is searched. As before, the handler term $castVote(Cand)$ unifies with the message term $castVote(liam)$, this time with $\theta' = \{Cand/liam\}$. θ' becomes the current unifier, and execution proceeds to the handler's body.

- `require` $candidate(Cand)$ is encountered. The current unifier θ' is applied, giving $candidate(liam)$. As $candidate(liam)$ is not in the knowledge base, SLDNF* fails to prove the goal. A `require_error` is thrown, and the phase ends with this exception.

The use of `require` is appropriate here because it is the end user's responsibility to specify a valid candidate to vote for.

Example 2

$$\begin{aligned}&\texttt{on}\ donate\\&\quad\texttt{if}\ value(Value), Value >= 1000\\&\quad\texttt{do}\ \texttt{assert}(gold_donor)\\[1em]&\texttt{on}\ donate\\&\quad\texttt{if}\ value(Value), Value < 1000\\&\quad\texttt{do}\ \texttt{assert}(standard_donor)\end{aligned}$$

This example contains two message handlers, an empty KB_s and an empty KB_d.

Consider a message with term *donate* and attached value of 1500 tokens of cryptocurrency. Execution proceeds as follows:

- *MH* is searched. Both handlers have matching handler terms and the top one is considered first. As the terms are constants, the unifier is empty in this case.
- `if` $value(Value), Value >= 1000$ is encountered. The condition is submitted to SLDNF*. Since SLDNF* employs a left-to-right subgoal literal selection function, $value(Value)$ is resolved first. Recall that *value* is a special context predicate which contains the amount of money sent with the message; hence, the fact is found in KB_c and the variable $Value$ is unified with 1500. Then since $1500 \geq 1000$, the entire `if` clause succeeds.
- `do assert`$(gold_donor)$ is reached, and the phase terminates with `assert`$(gold_donor)$. Note that the second handler is not executed.

Now consider the same message but with a value of 50.

- *MH* is searched. Again, the first handler has a matching handler term so its body is entered.
- `if` $value(Value), Value >= 1000$ is encountered. The condition is submitted to SLDNF*. This time, the condition is false. The rules for `if` dictate that execution is to backtrack to the previous choice point which, since this was the first body clause, goes back to message handler selection.
- The interpreter tries to find another message handler with a matching handler term. Only the second handler remains, which does unify. Execution enters its body.
- `if` $value(Value), Value < 1000$ is encountered. The condition succeeds.
- `do assert`$(standard_donor)$ is reached. The phase terminates with `assert`$(standard_donor)$.

In this example, `if` must be used to enable backtracking. Using `require` would not achieve the same effect: the first handler would throw an exception if $Value < 1000$ and execution would never reach the second handler.

Phase 2

Recall that if successful, the first phase returns some $action \in Actions$, which is obtained by applying some unification to the action of the first matching message handler. This action or composition of actions tells the second phase how to update the state of the program's dynamic knowledge base (in the case of `assert` and `retract`) and balances (in the case of `transfer`). Given some initial state, this phase can be thought of as returning either the final state or an exception.

Let the state be represented by a configuration $\langle KB_d, \sigma_b \rangle$, where KB_d is the smart contract's dynamic KB and σ_b is a mapping from addresses in *Addr* to integer balances in $\mathbb{N}$. We will use the notation $\sigma_b[a]$ to denote the value obtained by looking up the mapping σ_b at the address $a \in Addr$, and the notation $\sigma_b[a \mapsto n]$ to denote the new mapping obtained after assigning the value at a to n. Note that all addresses are mapped to 0 by default. Furthermore, let $caddr \in Addr$ to be the current smart contract's address.

The big-step (natural) operational semantics of *Actions* is given by transitions of the form

$$\langle action, KB_d, \sigma_b \rangle \Downarrow \langle KB'_d, \sigma'_b \rangle \mid \texttt{error}$$

from initial state to final state or exception.

$$\textsc{assert.success}\ \frac{\neg var(Term) \qquad KB'_d = KB_d \cdot \{Term\}}{\langle \texttt{assert}(Term), KB_d, \sigma_b\rangle \Downarrow \langle KB'_d, \sigma_b\rangle}$$

$$\textsc{assert.failure}\ \frac{var(Term)}{\langle \texttt{assert}(Term), KB_d, \sigma_b\rangle \Downarrow \texttt{error}}$$

An `assert` adds $Term$ as a fact to the dynamic KB. The operator $\cdot$ denotes an append to the end (recall that KB_d is ordered).

$Term$ must not be a variable; the `assert` fails if this is the case. We use the predicate $var(Term)$ to denote this condition. It is important to remember that the *action* evaluated in this phase of message interpretation has already been subjected to the unifications from the first phase. Consequently, it is acceptable for a message handler to be written with an `assert` term that is syntactically a variable; as long as the variable becomes unified with a constant or compound term before the first phase reaches the `do` clause, it will be received by the second phase as the instantiated term, and the `assert` will succeed.

Although $Term$ cannot be a variable, it need not be ground. For example, the term `has_restriction (james, X)` is legal. The evaluation of the `assert` itself makes no semantic interpretation of the $Term$ and appends it to the dynamic KB as is. The result is that the dynamic KB ascribes an implicit universal quantification to the variables. The action `assert(has_restriction(james, X))` can be read to say "make all restrictions apply to James", for example.

Note that `assert` always succeeds if $Term$ is instantiated, regardless of whether or not $Term$ is already present in KB_d. This is the same behaviour as in Prolog's `assertz`.

$$\textsc{retract.success}\ \frac{\neg var(Term) \qquad KB'_d = \{F \mid F \in KB_d \wedge \neg\exists\theta.[F\theta = Term\theta]\}}{\langle \texttt{retract}(Term), KB_d, \sigma_b\rangle \Downarrow \langle KB'_d, \sigma_b\rangle}$$

$$\textsc{retract.failure}\ \frac{var(Term)}{\langle \texttt{retract}(Term), KB_d, \sigma_b\rangle \Downarrow \texttt{error}}$$

A `retract` removes from the dynamic KB all facts that unify with $Term$. The order of the remaining facts is left unchanged. $Term$ must not be a variable, but may be non-ground. `retract` always succeeds if $Term$ is instantiated, even if no facts are removed – this is as in Prolog.

$$\textsc{transfer.success}\ \frac{\begin{array}{c}\sigma_b[caddr] = n \qquad n \geq Value\\ \sigma'_b = \sigma_b[caddr \mapsto n - Value,\ ToAddress \mapsto \sigma_b[ToAddress] + Value]\end{array}}{\langle \texttt{transfer}(ToAddress, Value), KB_d, \sigma_b\rangle \Downarrow \langle KB_d, \sigma'_b\rangle}$$

$$\textsc{transfer.failure}\ \frac{\sigma_b[caddr] = n \qquad n < Value}{\langle \texttt{transfer}(ToAddress, Value), KB_d, \sigma_b\rangle \Downarrow \texttt{error}}$$

A `transfer` is successful if the smart contract's balance is enough to cover the $Value$. If this is the case, $Value$ is subtracted from the contract's balance and added to the balance of $ToAddress$. Otherwise, the transfer evaluates to an error.

$$\textsc{comp.success}\ \frac{\langle action_1, KB_d, \sigma_b\rangle \Downarrow \langle KB'_d, \sigma'_b\rangle \qquad \langle action_2, KB'_d, \sigma'_b\rangle \Downarrow \langle KB''_d, \sigma''_b\rangle}{\langle (action_1, action_2), KB_d, \sigma_b\rangle \Downarrow \langle KB''_d, \sigma''_b\rangle}$$

$$\textsc{comp.failure}_1\ \frac{\langle action_1, KB_d, \sigma_b\rangle \Downarrow \texttt{error}}{\langle (action_1, action_2), KB_d, \sigma_b\rangle \Downarrow \texttt{error}}$$

$$\textsc{comp.failure}_2\ \frac{\langle action_1, KB_d, \sigma_b\rangle \Downarrow \langle KB'_d, \sigma'_b\rangle \qquad \langle action_2, KB'_d, \sigma'_b\rangle \Downarrow \texttt{error}}{\langle (action_1, action_2), KB_d, \sigma_b\rangle \Downarrow \texttt{error}}$$

A composite is executed in sequence. If any action should evaluate to an error, the entire composite evaluates to the error and subsequent actions are not considered.

4.3 Blockchain Specification

This section defines the Chainlog Platform's important data structures necessary to meet the previously discussed requirements: the address format, the authorisation scheme, the query and transaction definitions and the block structure. A specification of the blockchain's state machine is then given in a spirit similar to the Ethereum yellow paper [14].

4.3.1 Accounts, Addresses and Authentication

The Chainlog Platform, like Ethereum, has two account types. One is for external users; the other is for smart contracts. Public key cryptography is employed to realise identification and authentication.

An external user account is determined by a private and public keypair. The 33-byte public key is derived from the 32-byte private key using the secp256k1 Elliptic Curve Cryptography scheme. The user's address is then a 20-byte quantity obtained from the public key by concatenating it to the SHA-256 hash of a unique string and truncating the first 20 bytes[1]. For the purposes of identification, the address is formatted as a Bech32[2] string with the prefix `chainlog`. An example of an external user address is given below:

```
chainlog1cks45pqupl4g88e2axcvgemf23jrcg356h5xvh
```

Smart contracts do not have a cryptographic keypair. A contract's address is generated during its creation and is derived from the address of its creator. Specifically, the contract address is a 32-byte quantity equal to the SHA-256 hash of the creator address' raw bytes concatenated with the creator's 8-byte sequence number (the sequence number is the total number of transactions the creator has sent – it is used so that multiple contracts deployed by the same creator are given different addresses). As with external user addresses, contract addresses are formatted to their user-facing Bech32 string.

The set of addresses *Addr* is the set of all valid Bech32 address strings derivable by the two aforementioned schemes. This includes both user addresses and contract addresses. We shall also denote *CAddr* be the set of all valid Bech32 smart contract addresses.

Authentication is achieved by the use of cryptographic signatures. When an external user sends a transaction, they attach a signature generated from their private key (using secp256k1). The Chainlog platform is able to verify the signature against the user's public key.

[1] The address generation algorithm is defined by ADR-28, details of which can be found at https://github.com/cosmos/cosmos-sdk/blob/main/docs/architecture/adr-028-public-key-addresses.md

[2] See https://en.bitcoin.it/wiki/Bech32

4.3.2 Queries

A Chainlog blockchain query Q is a pair $\langle type, data \rangle$. The *type* identifies which category of query it is. The *data* is a payload containing additional arguments. From a user's perspective, a query can be thought of as a function call where *type* is the name of the function and *data* are the arguments. Thus, the notation `type(data_1, ..., data_n)` will be used to describe their format.

The Chainlog Platform supports, at minimum, the following query types.

`balance(address)`: Retrieves the cryptocurrency balance of any user or contract.

- Parameters:
  - `address`: the user or contract address, in $Addr$.
- Returns:
  - `balance`: the balance in $\mathbb{N}$.

`contractCode(contractAddress)`: Retrieves the code of the smart contract with the given address, including its dynamic KB. By making the code publicly queryable, the Chainlog Platform ensures the contractual terms are transparent to all parties.

- Parameters:
  - `contractAddress`: the contract address in $CAddr$.
- Returns:
  - `contract`: the smart contract program triple $\langle KB_s, KB_d, MH \rangle$

`queryContract(contractAddress, query, nDerivations)`: Submits a query to the knowledge base of the smart contract with the given address.

- Parameters:
  - `contractAddress`: the contract address in $CAddr$.
  - `query`: the logic goal to be submitted, as a conjunction of literals.
  - `nDerivations`: the maximum number of solutions to find.
- Returns:
  - `successful`: boolean indicating whether the query succeeded.
  - `derivations`: a list of derivations. Each element is a set of unifications made in a single derivation to the query, e.g. $\{X/john, Y/smith\}$.

4.3.3 Transactions

A transaction T is a pair $\langle msgs, sig \rangle$, where $msgs$ is a list of operations to be performed and sig is a wrapper around a cryptographic signature.

The transaction signature sig is a pair of the form $\langle signature, sequence \rangle$ where $signature$ is a secp256k1 signature and $sequence$ is a nonce.

The messages of $msgs$ each take the form $\langle type, data \rangle$ and, like queries, can be thought of as a function call `type(data_1, ..., data_n)`. These messages are of a broader notion than those processed by the Chainlog language interpreter in previous section – messages that invoke smart contract execution are one type of msg. As defined below, there are a total of three msg types the Chainlog platform supports.

`send(fromAddress, toAddress, amount)`: Sends cryptocurrency from one address to another.

- Parameters:
  - `fromAddress`: the user or contract address to send from, in $Addr$.
  - `toAddress`: the user or contract address to send to, in $Addr$.
  - `amount`: the amount to be sent in $\mathbb{N}$.
- Returns: (empty).

`createContract(creator, code, value)`: Deploys a new contract to the Chainlog platform.

- Parameters:

- `creator`: the address of the creator in $Addr$.
- `code`: the source code of the smart contract $KB_s \cup MH$ as a string.
- `value`: the initial amount of cryptocurrency to fund the smart contract with, in $\mathbb{N}$.

- Returns:
  - `contractAddress`: the address generated for the newly created smart contract, in $CAddr$.

`callContract(sender, contractAddress, value, messageTerm)`

- Submits a message term to a smart contract, attaching an amount of cryptocurrency.
- Parameters:
  - `sender`: the address of the sender in $Addr$.
  - `contractAddress`: the address of the smart contract in $CAddr$.
  - `value`: the amount of cryptocurrency to send with this message, in $\mathbb{N}$.
  - `messageTerm`: the message term, a single ground logic term.
- Returns: (empty).

4.3.4 Blocks

The Chainlog Platform block is fairly standard. It consists of a header and body.

The body contains a list of transactions.

The header contains (at least) the following information:

- **Height**: an incremental block number.
- **Timestamp**: UNIX timestamp in seconds. It is calculated to be the median time the network's validators approved the previous block. For the scope of this report, it is enough to know that it is Byzantine Fault Tolerant and deterministic. This is used by smart contracts as a source of up-to-date time.
- **Transactions Hash**: Merkle root of the hashed body transactions.
- **Application Hash**: hash of the application state. This is the hash of the σ obtained after execution of the previous block's transactions. The inclusion of this field allows light clients to verify state data quickly without the need to traverse the entire history of blocks.
- **Hash of Previous Block**: hash of the previous block's header.

4.3.5 State Machine Specification

Given the current blockchain application state σ and a transaction T, the Chainlog Platform's state transition function *apply* computes a post-transactional state σ':

$$\sigma' = apply(\sigma, T)$$

We are now ready to define what the state and transition function are.

The Chainlog application state σ is a mapping of mappings. Each contained mapping is a substate which corresponds to one application concern. In this work, we discuss three of these substates $\sigma_a, \sigma_b, \sigma_c$, where:

- σ_a is the *accounts mapping*, from $Addr$ to pairs of $\langle pubKey, sequence\rangle$.
- σ_b is the *balances mapping*, from $Addr$ to $\mathbb{N}$.
- σ_c is the *contracts mapping*, from $CAddr$ to program triples $\langle KB_s, KB_d, MH\rangle$.

The balances mapping and contracts mapping are straightforward. The accounts mapping includes information used for verification: *pubKey* is an account's public key and *sequence* is the number of transactions sent by the account. Some notational conventions are adopted. The mapping notation $\sigma_b[addr]$ and $\sigma_b[addr \mapsto 0]$ will be the same as that of the operational semantics section. For convenience, we shall represent the application state σ as a tuple $\langle \sigma_a, \sigma_b, \sigma_c\rangle$.

A definition of the state transition function *apply* now follows. Let

$$\sigma = \langle \sigma_a, \sigma_b, \sigma_c \rangle$$
$$T = \langle msgs, sig \rangle$$

be the current state and the transaction respectively. The transition function first performs some verification checks on the transaction's signature. Recall that *sig* is a pair:

$$sig = \langle signature, sequenceT \rangle$$

The transaction sender $saddr \in Addr$ is obtained from *signature*. This allows the lookup of the sender's account:

$$\sigma_a[saddr] = \langle pubKey, sequenceA \rangle$$

Then, the *signature* is verified against the sender's public key *pubKey*. In addition, the transaction's sequence number *sequenceT* must match the account's sequence number *sequenceA*; this is to protect against double-spending attacks. If these checks fail, the transaction is rejected.

Otherwise, the account's sequence number is incremented by one. Formally, an intermediate state σ_0 is derived from the initial state σ:

$$\sigma_0 = \langle \sigma_{a_0}, \sigma_b, \sigma_c \rangle \qquad \text{where}$$
$$\sigma_{a_0} = \sigma_a[saddr \mapsto \langle pubKey, sequenceA + 1 \rangle]$$

With the transaction validated and the sender authenticated, the messages in *msgs* may be processed. Let the individual messages be referred to as $msg_1, msg_2, ..., msg_n$. It is important that these are executed sequentially and in the given order, as the final result may depend on it. Each message either results in some update to state or throws an exception. Assuming all messages succeed, then we have:

$$\sigma_1 = exec(\sigma_0, msg_1)$$
$$\sigma_2 = exec(\sigma_1, msg_2)$$
$$\vdots$$
$$\sigma_n = exec(\sigma_{n-1}, msg_n)$$

where *exec* denotes the execution of a single message (to be defined shortly). In the absence of exceptions, the final post-transactional state σ' is this σ_n.

If any such *exec* should result in an exception, the entire transaction is regarded to have failed. All changes to state are reverted, with the exception of the increment to the sender account's sequence number. Thus, the post-transactional state σ' becomes σ_0.

In summary:

$$\sigma' = \begin{cases} \sigma_0 & \text{if } exec(\sigma_{i-1}, msg_i) = \texttt{error} \text{ for some } i \\ \sigma_n & \text{otherwise} \end{cases}$$
$$\text{where } \sigma_n = exec(...exec(exec(\sigma_0, msg_1), msg_2)..., msg_n)$$

The definition of *exec* depends on the type of the message; the three cases are given below. Given the current intermediate state σ_i and message *msg*, they each compute either the next intermediate state σ_{i+1} or the exception term `error`.

Execution of `send`

Let the current intermediate state and message be given as below.

$$\sigma_i = \langle \sigma_{a_i}, \sigma_{b_i}, \sigma_{c_i} \rangle$$
$$msg = \texttt{send(fromAddress, toAddress, amount)}$$

A `send` message results in an error if the balance of `fromAddress` is insufficient. Otherwise, the next intermediate state σ_{i+1} is the result of performing the transfer of `amount` in the balance mapping:

$$exec(\sigma_i, msg) = \begin{cases} \texttt{error} & \text{if } \sigma_{b_i}[\texttt{fromAddress}] < \texttt{amount} \\ \langle \sigma_{a_i}, \sigma_{b_{i+1}}, \sigma_{c_i} \rangle & \text{otherwise} \end{cases}$$

$$\text{where } \sigma_{b_{i+1}} = \sigma_{b_i}[\texttt{fromAddress} \mapsto \sigma_{b_i}[\texttt{fromAddress}] - \texttt{amount}, \\ \texttt{toAddress} \mapsto \sigma_{b_i}[\texttt{toAddress}] + \texttt{amount}]$$

Execution of `createContract`

Let the current intermediate state and message be given as below.

$$\sigma_i = \langle \sigma_{a_i}, \sigma_{b_i}, \sigma_{c_i} \rangle$$
$$msg = \texttt{createContract(creator, code, value)}$$

Furthermore, let us extract the creator's account and the provided source code's static KB and handler set:

$$\sigma_{a_i}[\texttt{creator}] = \langle pubKey, sequence \rangle$$
$$\texttt{code} = KB_s \cdot MH$$

A `createContract` message can fail in two cases: if the `code` of the smart contract exceeds the Chainlog platform's size limit of 65,536 bytes, or if the sender lacks the balance to fund the contract. The contract size limit exists to prevent denial of service attacks: giant programs cause a disproportional amount of effort for the blockchain network's nodes in, for example, reading and writing to disk and loading the code into the logic interpreter.

If the message passes the above checks, the following is performed:

- The contract's address is generated from the creator's address and sequence.
- The contract's account is initialised in the accounts mapping with a null public key and zero sequence.
- The contract's program triple is initialised in the contracts mapping with the provided KB_s and MH, and an empty KB_d.
- The contract is endowed its initial fund, financed from the creator's balance.

Formally:

$$exec(\sigma_i, msg) = \begin{cases} \texttt{error} & \text{if } ||\texttt{code}|| > 65536 \lor \sigma_{b_i}[\texttt{creator}] < \texttt{value} \\ \langle \sigma_{a_{i+1}}, \sigma_{b_{i+1}}, \sigma_{c_{i+1}} \rangle & \text{otherwise} \end{cases}$$

where:

$$\sigma_{a_{i+1}} = \sigma_{a_i}[caddr \mapsto acc]$$
$$\sigma_{b_{i+1}} = \sigma_{b_i}[\texttt{creator} \mapsto \sigma_{b_i}[\texttt{creator}] - \texttt{value},\ caddr \mapsto \texttt{value}]$$
$$\sigma_{c_{i+1}} = \sigma_{c_i}[caddr \mapsto P]$$

$$caddr = \texttt{bech32}(\texttt{sha256}(\texttt{bytes}(\texttt{creator}) \mathbin{+\!\!+} \texttt{bytes}(sequence)))$$
$$acc = \langle \texttt{null}, 0 \rangle$$
$$P = \langle KB_s, \emptyset, MH \rangle$$

Execution of `callContract`

Let the current intermediate state and message be given as below.

$$\sigma_i = \langle \sigma_{a_i}, \sigma_{b_i}, \sigma_{c_i} \rangle$$
$$msg = \texttt{callContract(sender, contractAddress, value, messageTerm)}$$

Furthermore, let $blockTimestamp$ be the timestamp of the current block.

A `callContract` message can fail in a number of cases: if there is no contract at the given address, if the sender lacks the balance to pay the specified value, or if the execution of the smart contract itself results in an exception.

Only the first two causes of error can be determined upfront. If they fail, the execution evaluates to an error, and we are done. Otherwise, we obtain a checkpoint state σ'_i after transferring the value from the sender to the contract:

$$\sigma'_i = \langle \sigma_{a_i}, \sigma'_{b_i}, \sigma_{c_i} \rangle \qquad \text{where}$$
$$\sigma'_{b_i} = \sigma_{b_i}[\texttt{sender} \mapsto \sigma_{b_i}[\texttt{sender}] - \texttt{value},\ \texttt{contractAddress} \mapsto \sigma_{b_i}[\texttt{contractAddress}] + \texttt{value}]$$

Let us extract from the state the (now known to be present) program of the contract:

$$\sigma_{c_i}[\texttt{contractAddress}] = \langle KB_s, KB_d, MH \rangle$$

The system now prepares for the execution of the smart contract. The context knowledge base KB_c is generated and populated with the four facts:

$$\begin{aligned} KB_c = \{&sender(S), \\ &value(V), \\ &time(T), \\ &balance(B)\} \end{aligned}$$
where:
$$\begin{aligned} S &= \texttt{sender} \\ V &= \texttt{value} \\ T &= blockTimestamp \\ B &= \sigma'_{b_i}[\texttt{contractAddress}] \end{aligned}$$

The first phase of the message interpreter is invoked. This procedure is as defined in the operational semantics section and will be denoted by the function $resolveMsg$ on the message term and combined program and knowledge bases:

$$result_1 = resolveMsg(\texttt{messageTerm}, P, KB_l, KB_c)$$

If $result_1 = \texttt{error}$ (for example, a `require_error` or `match_error`), then $exec$ for this message returns with the error. Recall that this results in the state being reverted; therefore, the transfer of value from the sender is refunded.

Otherwise, $result_1 = action$ for some $action \in Actions$. The second phase of the message interpreter evaluates $action$ with the contract's KB_d and the checkpoint state's balances mapping σ'_{b_i} in accordance with the operational semantics of actions:

$$\langle action, KB_d, \sigma'_{b_i} \rangle \Downarrow result_2$$

If $result_2 = \texttt{error}$, then $exec$ for this message returns with the error. Otherwise, $result_2$ is a configuration $\langle KB_d{}', \sigma''_{b_i} \rangle$ with a new dynamic KB and balances mapping. Execution of this `callContract` message is deemed to succeed, and the new intermediate state σ_{i+1} is the checkpoint state updated with this new dynamic KB and balances mapping:

$$\sigma_{i+1} = \langle \sigma_{a_i}, \sigma_{b_{i+1}}, \sigma_{c_{i+1}} \rangle$$

where:

$$\begin{aligned}\sigma_{b_{i+1}} &= \sigma''_b \\ \sigma_{c_{i+1}} &= \sigma_{c_i}[\texttt{contractAddress} \mapsto P'] \\ P' &= \langle KB_s, KB'_d, MH\rangle\end{aligned}$$

In summary:

$$exec(\sigma_i, msg) = \begin{cases} \texttt{error} & \text{if } Failure \\ \langle \sigma_{a_i}, \sigma_{b_{i+1}}, \sigma_{c_{i+1}}\rangle & \text{otherwise} \end{cases}$$

$$\begin{aligned}\text{where } Failure \iff\; & \texttt{contractAddress} \notin \text{dom}(\sigma_{c_i}) \\ & \vee\, \sigma_{b_i}[\texttt{sender}] < \texttt{value} \\ & \vee\, resolveMsg(\texttt{msgTerm}, P, KB_l, KB_c) = \texttt{error} \\ & \vee\, \langle action, KB_d, \sigma'_{b_i}\rangle \Downarrow \texttt{error}\end{aligned}$$

4.4 Summary and Extensions

The Chainlog language combines logic and procedural programming in a way that manifests the strengths of both frameworks. Its programs preserve the high-level, specification-like declarations of logic clauses, complemented by an intuitive mechanism to render instructions through imperative message handlers.

This is made operational by the Chainlog Platform, a full-fledged, blockchain-based smart contract platform which embeds a self-contained cryptocurrency and facilitates the deployment, storage and execution of Chainlog programs.

Together, the Chainlog language and platform enable the representation and execution of complex smart contracts in a trustless, decentralised and transparent environment.

There is nevertheless scope to expand on the current design, and a number of future extensions to both the language and the platform have been considered. We have alluded to the formalising of a pricing and gas model. The execution of a goal could associate a fixed gas cost with each inference made, which would place a deterministic limit on the amount of computation performed by the query interpreter. In addition, the actions of the message handlers could each be ascribed an appropriate cost of processing. An `assert`, for example, might incur a relatively high cost, as adding the new fact to the dynamic KB demands more of the blockchain's storage.

More generally, it is easy to equip smart contracts with new capabilities through the addition of new types of actions. For example, we conceive an action to emit logs in a similar manner to Solidity's `emit` instruction. An ambitious extension is to enable contracts to call to other Chainlog smart contracts on the platform. In fact, one of the reasons Chainlog's message interpretation procedure is split into two phases is to future-proof for this possibility, as it gives the flexibility for the second phase to batch together or reorder actions. This could prevent the possibility of the infamous reentrancy attack [23] entirely.

Chapter 5

Implementation

The previous chapter presented a formal design and specification for our logic-based language and blockchain. In this chapter, we turn theory into practice. The sections that follow showcase the two functional accomplishments of this work: the implementation of the Chainlog Language and the Chainlog Platform.

5.1 Language Implementation

The Chainlog language implementation is an amalgamation of components and tools which together facilitate the creation and execution of Chainlog smart contracts. Included are the following elements:

The Chainlog Interpreter : The core implementation of the query and message interpreter, written in Prolog.

Go-Chainlog : A Go library and API for creating, manipulating and invoking Chainlog interpreter instances.

The Library : The implementation of several predicates of the Chainlog standard library KB_l, in Prolog.

gochl : An interactive command-line Chainlog interpreter and message simulation engine.

The Testing framework : A set of utilities for unit testing Chainlog programs.

Syntax Highlighters : A Vim syntax file and LaTeX language definition to support syntax highlighting in Vim and LaTeX respectively.

5.1.1 Core Interpreter

The core Chainlog interpreter is a Prolog metaprogram. It is a single file which implements the Chainlog syntax, defines a subset of KB_l consisting of the built-in predicates and provides two fundamental entrypoints:

```
chainlog_query(:Goal, +QueryCtx)
chainlog_msg(+MsgTerm, +MsgCtx, -ActionsList)
```

The first is the query interpreter and the second is the first phase of the message interpreter.

The core interpreter is designed to be portable. It is almost entirely written in ISO Prolog and implements as much of the Chainlog interpretation process as the ISO predicates allow. To interpret a Chainlog program, one starts any Prolog engine, consults the Chainlog interpreter and program to be interpreted and calls one of the above two predicates.

Prolog is chosen because it already implements several of the language requirements: it supports clauses with variables, features a unification algorithm, provides negation as failure and uses a backward chaining inference strategy. Chainlog inherits Prolog's unification and negation as failure mechanisms directly, but reifies the search strategy to institute the modifications needed to transform it into SLDNF*.

The Chainlog keywords `on`, `require`, `do`, etc. are implemented using Prolog's support for defining custom operators. Prolog operators can only be unary or binary, which has led to the need for some additional syntax to demarcate the clauses of message handlers. In particular, the colon operator `(:)/2` is a non-associative binary operator that separates the handler from the body, and the semicolon operator `(;)/2` is a right-associative operator that separates body clauses. This is why Chainlog programs in reality, such as those showcased in the evaluation section, appear to have more punctuation. Listing 5.1 shows an example of a message handler in this practical syntax. Notice the colon on line 1 and the semicolon on line 2.

```
on payRent:
  require value(V), V is 366;
  do      assert(rent_paid).
```

Listing 5.1: Chainlog Practical Syntax

The keywords `on`, `require` and `do` are unary. Operator precedence follows the following order:

$$\texttt{on} \prec \texttt{(:)} \prec \texttt{(;)} \prec \texttt{require} = \texttt{if} = \texttt{do} \prec \texttt{(,)}$$

Given these binding rules and the fact that operators are merely syntactic sugar for predicates and terms, the handler in listing 5.1 is therefore understood by the core interpreter as an `on/1` fact with the form shown in listing 5.2:

```
on(
  :(
    payRent,
    ;(
      require(
        ,(
          value(V),
          is(V, 366)
        )
      ),
      do(
        assert(rent_paid)
      )
    )
  )
).
```

Listing 5.2: Chainlog True Syntax Without Syntactic Sugar

A Chainlog developer writes the smart contract's source code $KB_s \cup MH$ together in one file with the `.chl` extension. The KB_s is written in Prolog-style syntax except that negation as failure is a unary operator `not` and disjunction is a binary operator `or`. The MH is written in the syntax shown earlier. Listing 5.3 shows a real example for remote patient monitoring.

```
% Static KB
monitoring_device('chainlog1g8zm5a7ax2lnjsd6v4r73ps4xm9q8cwu906a83').

abnormal(Patient) :-
  heart_rate(Patient, Rate),
  not between(60, 100, Rate).
abnormal(Patient) :-
  blood_pressure(Patient, blood_pressure(Systolic, Diastolic)),
  (Systolic > 120
   or Diastolic > 80).


% Message Handler
on addPatientData(Patient, HeartRate, blood_pressure(Systolic, Diastolic)):
  require sender(Sender), monitoring_device(Sender);
  do      assert(heart_rate(Patient, HeartRate)),
```

```
17            assert(blood_pressure(Patient, blood_pressure(Systolic, Diastolic))).
```

Listing 5.3: Example Chainlog Program

Query Interpreter

The `chainlog_query(:Goal, +QueryCtx)` predicate implements the query interpreter. `Goal` is the goal to be submitted, a literal or conjunction of literals. `QueryCtx` represents the context knowledge base KB_c and is expected to be a compound term of the form `query_ctx(Time, Balance)`.

The procedure is essentially a decorator around a Prolog metainterpreter. The decorator sets up the query context KB_c and destroys it immediately after. The metainterpreter is a vanilla Prolog metainterpreter with three differences. First, it keeps a countdown of the depth and throws an exception when the countdown reaches zero. Second, it adds support for the Chainlog operators `not` and `or`. Third, its clauses are ordered so as to enforce the search of the knowledge bases in the required order $KB_l \prec KB_s \prec KB_c \prec KB_d$.

Message Interpreter

The `chainlog_msg(+MsgTerm, +MsgCtx, -ActionsList)` predicate performs the first phase of message interpretation. Given the message term `MsgTerm` and a message context `MsgCtx` of the form `msg_ctx (Sender, Value, Time, Balance)` representing KB_c, it searches for a matching message handler and unifies its actions with `ActionsList`.

It works by attempting to find a fact of the form `on (HandlerTerm: Body)` – recall that message handlers are just syntactic sugar for `on/1` facts – and unify `MsgTerm` with `HandlerTerm`. If such a handler is found, it delegates to a helper procedure to execute the `Body` with two decorators.

The first decorator is responsible for setting up the message context knowledge base from the `MsgCtx` and destroying it afterwards. The second decorator catches any `require_error` exceptions that occur during the body's execution. If an error is caught, it attaches the message term and context to the error and rethrows it. Its purpose is simply to make exception messages more descriptive for the user.

Execution of the message body involves a recursive predicate that goes through each of its body clauses until it reaches the base case that is the `do` clause. Any `if` and `require` conditions encountered during the recursion are submitted to a variant of the query interpreter without the context decorator (since the message context already includes the time and balance). If the `do` clause is reached, its actions are unrolled into a list and returned.

Because the predicate is recursive, the search tree of the body's execution amounts to one long branch. Hence, all unifications made in the process accumulate as the procedure recurses. The "current unifier" is maintained by Prolog as part of its resolution algorithm.

Only the first phase of message interpretation is performed by the Prolog component. We will see that this is because the balances mapping is maintained outside the program. Prolog lacks a portable way to convey the movement of money that should result from a `transfer` action to this external mapping.

5.1.2 Go-Chainlog

Go-Chainlog serves two purposes. The first is to complement the core Prolog interpreter in areas that portable Prolog fails to express. The second is to provide a library around this interpreter to invoke it from the Go language. Go-Chainlog forms the basis of the `gochl` interactive interpreter and the testing framework's test runner. Furthermore, it is this component that the Chainlog Platform implementation embeds and invokes to execute smart contracts.

At the heart of Go-Chainlog is a Prolog engine. A third-party engine ichiban/prolog[1] is used, which is a very basic and lightweight implementation of the language written in Go. Go-Chainlog's primary functionality is to spawn an instance of this Prolog engine, load the core Chainlog interpreter

[1]https://github.com/ichiban/prolog

into it, expose utilities for loading Chainlog programs and provide wrappers around `chainlog_query` and `chainlog_msg`. Its API insulates the Prolog interpreter from its users.

One key feature of the ichiban/prolog engine is that it supports sandboxing: it provides a Prolog interpreter stripped of all built-in predicates and operators, allowing one to selectively include only the built-ins they desire. This feature is used to realise the restrictions of the Chainlog language. Predicates that usually come built-in with Prolog such as `open`, `write`, etc. have been intentionally excluded in order to ensure the isolation of smart contract execution.

Some important types and functions of the Go-Chainlog public API are highlighted:

- **`type Interpreter`**: represents a single instance of a Chainlog interpreter. The functions prefixed with "`i.`" below are methods on this type.
- **`type QueryContext`**: a wrapper around a `query_ctx(Time, Balance)`. Represents the ephemeral context KB_c during a blockchain query.
- **`type MessageContext`**: a wrapper around a `msg_ctx(Sender, Value, Time, Balance)`. Represents the ephemeral context KB_c during a message call.
- **`type Action`**: an interface type representing a single, non-composite action.
- **`type QueryIterator`**: an iterator of potential solutions to a Chainlog query.
- **`func NewInterpreter()`**: creates and returns a new sandboxed `Interpreter` with the Chainlog library built-in.
- **`func i.ConsultWithDynamicKB()`**: loads a full Chainlog program triple $\langle KB_s, KB_d, MH\rangle$ into the interpreter.
- **`func i.QueryWithContext()`**: the query interpreter, a wrapper around the core interpreter's `chainlog_query`. It takes a goal and a `QueryContext` and returns a `QueryIterator` of derivations or an error.
- **`func i.Message()`**: the message interpreter, a wrapper around the core interpreter's `chainlog_msg`. It takes a message term and a `MessageContext` and returns a list of `Action` or an error.
- **`func i.Assert()`**: implements the `assert` action. It essentially performs a Prolog `assertz` on the dynamic KB.
- **`func i.Retract()`**: implements the `retract` action. It essentially performs a Prolog `retractall` on the dynamic KB.
- **`func i.GetDynamicKB()`**: extracts and returns the current program's dynamic KB.

Notice that Go-Chainlog still does not implement the second phase of message interpretation. The task of interpreting the actions returned by `i.Message()` is deferred to the caller. It does, however, provide methods for executing single `assert` and `retract` actions.

The Go-Chainlog API is a published Go module. It is able to be fetched by Go's package management tools. We will see that the Chainlog Platform implementation in the next section imports Go-Chainlog and uses the methods above to execute smart contracts.

5.1.3 Other Tools

A brief mention is given to the surrounding development tools that have been included in the language implementation.

The interactive interpreter `gochl` is a command line interface to Go-Chainlog. It provides a quick means of experimenting with Chainlog programs. Users can load their smart contracts from files and submit both queries and messages. `gochl` simulates a customisable blockchain environment, which allows the user to configure the query and message contexts and inspect the changes to state caused by the execution of an action. This makes it convenient for developers to test that their contracts behave as expected across a range of different inputs and contextual conditions.

For more formal testing of Chainlog smart contracts, the implementation also provides a unit testing framework. This includes a test runner written in Go and a library of Prolog predicates for making test assertions (e.g., a given query succeeds, a given message produces a certain set of actions, a given exception is thrown, etc.) and mocking query and message contexts.

5.2 Blockchain Implementation

We present a realisation of the Chainlog Platform specification implemented using the Cosmos SDK and written in Go. This section describes its architecture and operation.

5.2.1 Architecture

The Chainlog blockchain is a network of nodes connected in a peer-to-peer fashion. Nodes come in three types: full nodes, validator nodes and light clients. For the scope of this report, we shall restrict our attention to full nodes only.

A participant wishing to join the network executes the full node daemon. This is a binary which bundles together the Chainlog Platform application with an instance of Tendermint Core. Figure 5.1 presents a system architecture diagram of its various components.

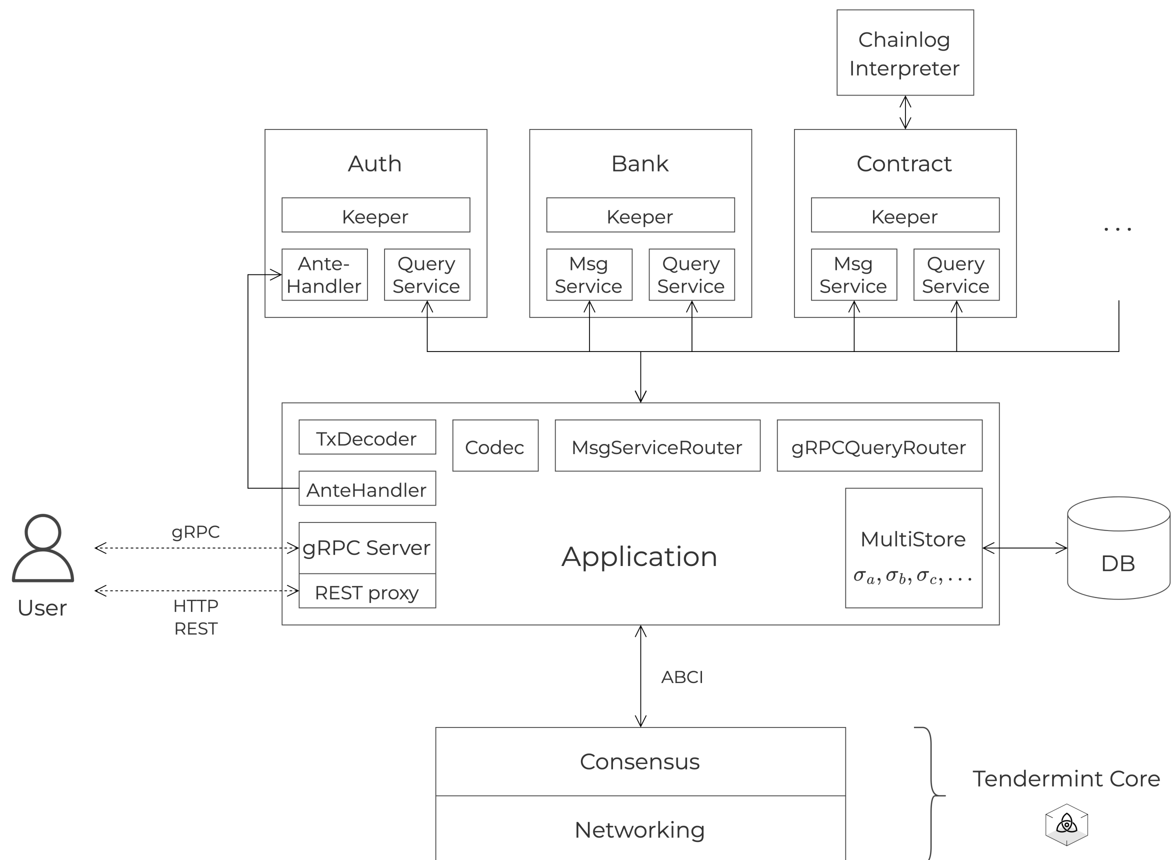

Figure 5.1: System Architecture Diagram of Full Node Daemon

From bottom to top, this includes:

Networking : Handles peer-to-peer communication with other nodes, including the relaying of transactions and blocks.

Consensus : Handles consensus; establishes a deterministic ordering of transactions by means of a Byzantine Fault Tolerant Proof-of-Stake mechanism.

Application : The core of the node's application level. This is the component that implements the Application Blockchain Interface (ABCI). It is itself composed of several important subcomponents:

gRPC Server : Services requests from external clients (e.g., wallets and browsers) on port 9090. gRPC is a modern Remote Procedure Call (RPC) framework by Google that is widely used in industry to bind microservices and connect client devices to server backends [55]. A full node also exposes a reverse-proxy REST server on port 1317, which routes HTTP requests to the gRPC server.

Reference to AnteHandler : The AnteHandler is a set of decorators used for checking transaction validity and account bookkeeping. It is responsible for verifying transaction signatures and checking and incrementing the account sequence number.

TxDecoder : When transactions are received from Tendermint Core through the ABCI, they are still in raw byte form. The TxDecoder is used to unmarshal the bytes to an application-level interface type.

Codec : Used for marshalling and unmarshalling state data structures to and from bytes. Protocol Buffers (Protobuf) is used for the encoding of types – see the subsection below.

MsgServiceRouter : Maps blockchain messages to the correct module's Msg Server based on their types.

gRPCQueryRouter : Maps blockchain queries to the correct module's Query Server based on their types.

MultiStore : The implementation of the application state σ. The MultiStore is a store of key-value stores; each constituent key-value store represents one module's partition of the state. The mappings $\sigma_a, \sigma_b, \sigma_c$ are key-value stores belonging to the auth, bank and contract modules respectively.
Each key-value store is a mapping from bytes to bytes. The data structure used is a variant on an AVL+ tree called IAVL [56], which possesses several important properties that make it suitable for storing blockchain state. First, it is *snapshottable*: the tree is immutable; modifications produce a new tree; so, it is possible to access a previous snapshot of the state. The new tree is able to share nodes with its predecessor, which makes it easy to revert to a previous state. Second, a deterministic Merkle root hash of the tree is able to be computed. Third, the tree is self-balanced using a variant of AVL such that get and set operations are $\mathcal{O}(\log n)$.
The MultiStore is a hash map of key-value stores. Its hash is simply the Merkle root of the hashes of the contained IAVL stores ordered by name.

DB : A LevelDB database used to persist application state.

Modules : Where the application logic is implemented. Each module defines two Protobuf services: the Msg Service is a set of RPC methods for handling the subset of blockchain messages that concern the module, and the Query Service is a gRPC service to handle the module's subset of blockchain queries.
Each module also has an associated keeper which guards access to its partition of σ. For security reasons, modules cannot read from and write to the store directly; they must do so via the keeper. One module's keeper may be given a reference to another's keeper if it needs to access its state. This is a protection mechanism – it enables a module to define restrictions on the operations other modules are able to perform on its state.
The Chainlog Platform implementation consists of fourteen modules in total; it is not possible to cover all fourteen in this report. Discussion will surround the most pertinent three: auth, bank and contract.

Auth : Manages accounts. The auth keeper holds a key to access the accounts key-value store σ_a and exposes methods for retrieving accounts by address and creating new accounts. Notice that this module does not have a Msg Service as it does not need to process any messages. Query Service for auth provides a method `account(address)` for retrieving an account by address – users can call this to view their sequence number. The AnteHandler also resides in the auth module as it needs to access account public keys and sequences. This module is provided by the Cosmos SDK.

Bank : Manages cryptocurrency balances. The bank keeper holds a key to access the balances key-value store σ_b and exposes methods for retrieving balances and performing transfers. Msg Service for bank provides the `send(fromAddress, toAddress, amount)` method. Query Service for bank provides the `balance(address)` method. This module is provided by the Cosmos SDK.

Contract : Manages Chainlog smart contracts and is responsible for all contract interaction. The contract keeper holds a key to access the contracts key-value store σ_c and also contains references to the auth keeper and bank keeper. This is because contract creation needs to create a new entry in the account store (managed by auth), and both contract creation and invocation need to transfer money in the balances store (managed by bank). Msg Service for contract provides the `createContract(creator, code, value)` and `callContract(sender, contractAddress, value, messageTerm)` methods. Query Service for contract provides the `contractCode(contractAddress)` and `queryContract(contractAddress, query, nDerivations)` methods.

Chainlog Interpreter : An instance of the Go-Chainlog implementation described in the previous section. The contract module calls to this when it needs to query or message a smart contract.

When the full node daemon is started, it begins by initialising its state machine. If the daemon has been executed before, the last known state is loaded from DB. Tendermint will perform a handshake with peer nodes and acquire the new blocks that have been added to the blockchain since the node was last run. The local state will then be synced up to the latest canonical state by executing the blocks in sequence.

5.2.2 Transaction Lifecycle

Let us take a deeper look at the steps involved in processing a blockchain transaction.

A user starts by constructing a transaction with one or more messages and signing it with their private key and sequence number. They then send the transaction to a full node, either by using a gRPC client to connect to the full node's gRPC server or establishing a HTTP connection to the node's REST proxy. One can envisage this to be abstracted away by a friendly user interface and wallet.

When the application receives the transaction, it is handed to the Tendermint consensus machinery.

Tendermint maintains a local mempool: an in-memory pool where transactions wait to be included in a block. Before adding the transaction to the mempool, Tendermint offers the application the opportunity to reject the transaction if it is invalid. It sends a `CheckTx` ABCI message to the application containing the transaction bytes and awaits a positive or negative response. This is essentially an optimisation step to guard the mempool from spam transactions.

When the Chainlog Platform application receives the `CheckTx` message, it unmarshals the raw transaction bytes using the TxDecoder and extracts the list of messages. For each message, a number of stateless checks are performed, such as verifying it is well-formed, addresses are valid, etc. The size of the code given in a `createContract` message is verified to be within the size limit in this step. If all messages are valid, then the AnteHandler is invoked to perform stateful checks of the transaction's signature and sequence number. Should any of these stateless or stateful checks fail, the application will return a negative `CheckTx` response to Tendermint. Otherwise, a positive response is sent.

If Tendermint receives a negative `CheckTx` response, the transaction is discarded, and the lifecycle ends there. Let us instead assume the transaction is valid. It is added to the local mempool and relayed to peer nodes. At this point, the transaction sits in the nodes' mempools until it is ready to be included in a new block.

The consensus algorithm will eventually nominate a proposer among the network's validators to create a new block. Note that since Tendermint employs a Proof-of-Stake protocol, there is no

competition between nodes to create blocks as there is between miners in Proof-of-Work; a single validator is bestowed this privilege. This proposer will broadcast their new block to the network. We resume the transaction's lifecycle when each full node receives word of the proposed block containing it.

Upon receipt of the block, the Tendermint instance of each full node extracts the contained transactions and sends them to the application one at a time to be executed. Each transaction is packaged in a `DeliverTx` ABCI message. The order of transactions is important for determinism; thus, every full node delivers the transactions to the application in the precise order given by the proposer, and only after one transaction has been fully executed is the next transaction's `DeliverTx` sent.

When a `DeliverTx` is received by the Chainlog Platform application, the transaction is again unmarshalled by the TxDecoder and the messages are extracted. All stateless and stateful checks on the transaction are repeated; the AnteHandler executes again (the application state may have updated since the `CheckTx` was performed, which may render the transaction invalid). For each message, the application then does the following. It consults the MsgServiceRouter to determine which module's Msg Service to route the message to based on its type. It sends the message to the service, which routes it to the appropriate RPC method. It is in this RPC method that the message is actually executed.

While executing the message, it is expected that the RPC method will make calls to the module's keeper to read from and effect changes to state. It may also make calls to other modules' keepers for which the local keeper has a reference to. For example, the `createContract` message will call the auth keeper to create a new account for the contract and the bank keeper to endow its initial fund.

A message may result in an exception, which as the Chainlog Platform specification stipulates will cause the entire transaction to fail, and for changes to state to be reverted. The application immediately ceases message processing and returns a negative response to Tendermint. The failure of a transaction furthermore results in the rejection of the entire block. Let us, however, assume the case where all messages succeed. After the execution of the final message completes, the state transition function will have been fully applied and the MultiStore[2] updated to its post-transactional state σ'. The application returns a positive response to Tendermint.

Because the transactions and messages were executed in a well-defined order, and because the Chainlog Platform's state transition function is deterministic, every full node that executed the block has now computed the same new state σ_{t+1} (or all rejected the block). However, the network's canonical state is not updated yet – the block needs to be approved by the network's validators. It is enough to know for the purposes of this report that the validators attain consensus through Tendermint's protocol and a verdict is reached on whether to accept or reject the block. If the block is accepted, the Tendermint instance on each full node sends a `Commit` ABCI message to the application. This is to simultaneously inform the application that the changes to state should be finalised and to request the hash of the new application state σ_{t+1} so that it can be included in the header of the next block.

The Chainlog Platform application responds to the `Commit` by persisting the new state σ_{t+1} to the DB and computing and returning its Merkle root hash.

Finally, the new block is appended to the blockchain. The application state has now canonically transitioned to the new state σ_{t+1}. The lifecycle of the transaction ends here, but an immutable record of its occurrence remains in the blockchain forever.

5.2.3 Query Lifecycle

The lifecycle of a blockchain query is simpler than that of a transaction, due to the fact that it does not require consensus to be established.

[2]In reality, the post-transactional state is written to an isolated copy of the MultiStore which was derived by store branching during the `BeginBlock` ABCI call, received by the application just before the first `DeliverTx`. This is so that the branched store can (a) be cached for reads and (b) be discarded if state is to be reverted. Details of store branching, the `BeginBlock` ABCI message and a host of other ABCI messages have been omitted in this report for simplicity.

A user constructs a query and sends it to a full node. As with a transaction, this can be done through the node's gRPC server or REST proxy.

The query is processed purely at the application level. After it is received by the gRPC endpoint, the application looks up the gRPCQueryRouter to determine which module's Query Service the query should be sent to. Based on the type, it is routed to this service and to the correct query handler.

The query handler processes the query accordingly, using the keeper's keys to retrieve state as required, and returns a response. The response finds its way back to the user.

5.2.4 Protocol Buffers

Protocol Buffers or Protobuf is an open source, language-neutral data serialisation format for the storage and network communication of structured data [57]. To use Protobuf, a developer writes `.proto` files defining *messages*, which are record-like data structures with typed fields, and *services*, which are APIs containing RPC methods. The Protobuf compiler `protoc` is then able to auto-generate code in a number of target languages including C++, Java, Ruby, Dart and Go. Messages compile to complex data types complete with methods for getting and setting fields and for byte serialisation and deserialisation. Services compile to full gRPC server and client code.

The Chainlog Platform implementation uses Protobuf messages to define its state datatypes (e.g., contracts) as well as message and query parameter data. For example, listing 5.4 shows the Protobuf definition used for the parameter and return types of the `createContract(creator, code, value)` message.

```
message MsgCreateContract {
  string creator = 1;
  string code = 2;
  string value = 3;
}

message MsgCreateContractResponse {
  string contractAddress = 1;
}
```

Listing 5.4: Protobuf Message Definitions for `createContract`

Protobuf services are used to define the Msg Service and Query Service for each module. For example, listing 5.5 gives the contract module's Msg Service definition.

```
service Msg {
  rpc CreateContract(MsgCreateContract) returns (MsgCreateContractResponse);
  rpc CallContract(MsgCallContract) returns (MsgCallContractResponse);
}
```

Listing 5.5: Protobuf Service Definition for Contract Module Msg Service

The use of Protocol Buffers makes the marshalling and unmarshalling of data types and the writing of the RPC servers exceedingly simple, as `protoc` generates all the required code automatically.

5.2.5 Contract Representation

Recall that Chainlog smart contracts are triples $\langle KB_s, KB_d, MH \rangle$ stored in the contracts mapping σ_c by address, and this mapping is implemented as the key-value store that constitutes the contract module's partition of the MultiStore. New smart contracts are deployed by the `createContract` message, which is routed to the contract module's associated RPC method. The method receives the message parameters in a structure following that of `MsgCreateContract` in listing 5.4.

The contract key-value store is a mapping from bytes to bytes. Thus, it is necessary to marshal the triple $\langle KB_s, KB_d, MH \rangle$ to byte form. To aid this, a Protobuf message representing the contract triple has been defined, and is shown in listing 5.6. The `code` is the program's source code $KB_s \cup MH$. The `dynamicKb` is the string form of the program's dynamic knowledge base KB_d.

```
message SmartContract {
  string code = 1;
  string dynamicKb = 2;
}
```

Listing 5.6: Protobuf Message Definition for Smart Contract State

The Protobuf compiler `protoc` compiles this into a Go struct type definition called `SmartContract`, with methods to marshal and unmarshal it to and from bytes.

During a `createContract(creator, code, value)` message, a new `SmartContract` struct is initialised with the `code` taken directly from the `code` parameter of the message, and the `dynamicKb` assigned to the empty string. The `createContract` RPC method calls a setter method on the contract module's keeper to insert the struct into the contracts mapping. This keeper method invokes the application's codec to marshal the struct to bytes; the codec calls the Protobuf-generated marshal method. The bytes are put into the mapping.

After the contract is added to the store, the `code` is forevermore immutable. This is to ensure the terms of the contract cannot be altered maliciously. At any time, users can retrieve both the `code` and `dynamicKb` via the `contractCode` query. This is important to keep the terms of the contract visible and transparent.

5.2.6 Contract Execution

The processing of the `queryContract` query and the `callContract` message require the targeted Chainlog smart contract to be executed. The former submits the specified goal to the SLDNF* procedure with the contract's knowledge base; the latter invokes the Chainlog message interpreter with the specified message term. To facilitate both kinds of execution, the contract module imports the Go-Chainlog package.

In addition to all relevant validity checks and state lookups, the `queryContract(contractAddress, query, nDerivations)` query's RPC method:

- Calls `NewInterpreter` to spawn a new Chainlog interpreter instance with the library knowledge base KB_l loaded.
- Calls `ConsultWithDynamicKB` to load the full program triple.
- Initialises a `QueryContext` representing an ephemeral context knowledge base KB_c, with the time being the block timestamp and the balance being the result of looking up σ_b[`contractAddress`] via the bank keeper.
- Calls `QueryWithContext` to invoke the SLDNF* procedure, passing the supplied `query` and the created `QueryContext`. If successful, this yields a `QueryIterator` of potential derivations.
- Attempts to invoke `Next` on the iterator `nDerivations` times or until there are no more solutions to the goal.
- Returns all the unifications found.

If the `QueryWithContext` call or any `Next` call returns a non-nil error, the query fails and the error is reported to the user.

When a `callContract(sender, contractAddress, value, messageTerm)` message's RPC method comes to execute the contract, it:

- Calls `NewInterpreter` to spawn a new Chainlog interpreter instance with the library knowledge base KB_l loaded.
- Calls `ConsultWithDynamicKB` to load the full program triple.
- Initialises a `MessageContext` representing an ephemeral context knowledge base KB_c with the sender, value, time and balance.

- Calls `Message` to perform the first phase of message interpretation, passing the supplied `messageTerm` and the created `MessageContext`. If successful, this yields a list of `Actions`.
- Performs the second phase of message interpretation. The logic is implemented in the RPC method, not the Chainlog interpreter. For `assert` and `retract` actions, it calls the `Assert` and `Retract` methods on the interpreter appropriately. For `transfer` actions, it invokes the bank keeper to send the money.
- Calls `DynamicKBAsLogicProgram` to get the string form of the updated dynamic KB if there were any `assert` or `retract` actions. Afterward, the contract keeper is invoked to update the contract's dynamic KB in state to this new string.

Any errors resulting from the call to `Message` or any `transfer` action will be reported back to the transaction sender.

5.3 Summary and Extensions

We have seen in this chapter the implementation of the Chainlog language: an interpreter written using the metalogical capabilities of Prolog, complemented by a number of utilities in Go for interfacing and testing Chainlog smart contracts. We then detailed the Chainlog Platform implementation: a complete blockchain built from the ground up which actualises the Chainlog Platform specification.

The language interpreter's use of Prolog invites many interesting possibilities in terms of extensions. More augmentations to the query metainterpreter could be considered – a common one often seen in pedagogy is the explainable interpreter, which provides the proof tree for a query's derivation. This could be useful in helping parties to navigate complex contractual conditions, e.g. "John is in default of the agreement, because he has been found to breach article A, because he did X on date Y but did not do Z, ...". Since the Chainlog programs of our implementation are syntactic sugar over Prolog, it would also be easy to devise other Prolog metaprograms which read and process Chainlog contracts as data. This could be used to create tools for analysis and verification.

Chapter 6

Evaluation

This chapter seeks to evaluate the Chainlog design and implementation against the criteria of expressiveness and security. A showcase of the language by example forms the bulk of this analysis, aiming to illustrate its capacity to represent a broad range of contracts in a high-level, declarative manner. We then proceed to a discussion highlighting the security guarantees of the Chainlog Platform and demonstrating its robustness to attacks.

6.1 Demonstration of the Language

This section serves to showcase the expressiveness of the Chainlog language. We present smart contracts for four real-life use cases: crowdfunding, licensing, insurance and pharmacogenomics. In each case, we give both the Chainlog and Solidity representations and analyse the comparative advantages of our language.

All four contracts demonstrate Chainlog to be high-level and close to specification. In the crowdfunding and insurance examples, we argue that this leads to code that is more readable and comprehensible. The licence and pharmacogenomics examples show this to result in more error-free contracts. In addition, the crowdfunding and pharmacogenomics examples evidence the power of Chainlog's query mechanisms.

Several of the longer contracts have been abridged in this section. The full code for all examples is available in appendix A.

6.1.1 Crowdfunding

We first take a look at a simple crowdfunding contract, a common example for showcasing smart contract languages. The one presented here represents a single campaign. It has a beneficiary, a goal amount to reach and a deadline. Anyone is able to contribute to the campaign by sending cryptocurrency to the contract. If at the time of the deadline the goal has been reached, the beneficiary becomes able to claim the raised funds. Otherwise, the beneficiary receives nothing and the funders are able to collect a refund.

At any given moment, the smart contract will be in one of four states: fundraising, unsuccessful, successful or closed. The contract starts in the fundraising state and remains this way until the deadline. Contributions are accepted during this state. The unsuccessful state occurs when the goal is not reached by the deadline. It is in this state that the funders permitted a refund. The successful state occurs when the deadline is reached and the goal has been met or exceeded. After the beneficiary claims the funds during the successful state, the contract moves to the closed state, but the records of how much each funder contributed are kept in the smart contract for future reference. The idea is that this state is visible to participants so that they know when they are able contribute, claim funds or collect a refund.

Listing 6.1 presents our Chainlog implementation.

```
beneficiary('chainlog1n0tekhkd77ttvpac52jst8764l2qm952ghygvz').
deadline(1656633600).
goal(100000).

past_deadline :- time(Now), deadline(Deadline), Now > Deadline.
reached_goal :- total_raised(Total), goal(Goal), Total >= Goal.

state(fundraising)  :- not past_deadline.
state(unsuccessful) :- past_deadline, not reached_goal.
state(successful)   :- past_deadline, reached_goal, not claimed.
state(closed)       :- claimed.

total_raised(Total) :- sum_all(Contribution, funded(_, Contribution), Total).


on contribute:
  require state(fundraising);
  if      sender(Funder), value(Contribution);
  do      assert(funded(Funder, Contribution)).

on claimFunds:
  require state(successful);
  require sender(Sender), beneficiary(Sender);
  if      total_raised(Total);
  do      assert(claimed),
          transfer(Sender, Total).

on refund:
  require state(unsuccessful);
  if      sender(Sender),
          sum_all(Contribution, funded(Sender, Contribution), Total);
  do      retract(funded(Sender, _)),
          transfer(Sender, Total).
```

Listing 6.1: Chainlog Crowdfunding Contract

The logic-based approach allows us to define the four states and the total amount raised declaratively. By doing so, the meaning of each is easy to understand, and the program reads like a natural language specification: "the state is unsuccessful when past the deadline and the goal is not reached", "the total amount raised is the sum of all individual contributions"[1].

This is in contrast to the Solidity expression, which is given in listing 6.2. In the procedural paradigm, the state and total amount raised are variables whose values the programmer must maintain manually. The meaning of each becomes scattered across the different procedures of the program. For example, to obtain an understanding of what the value of `state` means, the reader needs to look at (a) its declaration on line 16 to see its initial value, (b) the `claimFunds` function to see that the closed state is assigned on line 31 and (c) the `updateState` function on lines 44 through 55 to see under what conditions the successful and unsuccessful states are set. As contracts grow in size and number of variables, the issue of comprehensibility only becomes compounded as the assignments to different variables become intertwined and the order in which they are executed affects their correctness.

```
pragma solidity ^0.8;

contract CrowdFunding {

    address public constant BENEFICIARY = 0xAa596dFfeA2f94668A2E667Ced218442cA3F10DE;
    uint public constant DEADLINE = 1656633600;
    uint public constant GOAL = 100 ether;

    enum State {
        FUNDRAISING,
        UNSUCCESSFUL,
        SUCCESSFUL,
        CLOSED
    }
```

[1] `sum_all(Template, Goal, Sum)` is a Chainlog library predicate equivalent to SWI-Prolog's `aggregate_all(sum (Template), Goal, Sum)`. It performs a `findall` and sums the results.

```
15
16      State public state = State.FUNDRAISING;
17      mapping(address => uint) public contributions;
18      uint public totalRaised;
19
20      function contribute() external payable {
21          updateState();
22          require(state == State.FUNDRAISING, "campaign not in fundraising state");
23          contributions[msg.sender] += msg.value;
24          totalRaised += msg.value;
25      }
26
27      function claimFunds() external {
28          updateState();
29          require(state == State.SUCCESSFUL, "campaign not in successful state");
30          require(msg.sender == BENEFICIARY, "caller is not the beneficiary");
31          state = State.CLOSED;
32          payable(msg.sender).transfer(totalRaised);
33      }
34
35      function refund() external {
36          updateState();
37          require(state == State.UNSUCCESSFUL, "campaign not in unsuccessful state");
38          uint contribution = contributions[msg.sender];
39          totalRaised -= contribution;
40          delete contributions[msg.sender];
41          payable(msg.sender).transfer(contribution);
42      }
43
44      function updateState() public {
45          if (state == State.CLOSED) {
46              return;
47          }
48          if (block.timestamp > DEADLINE) {
49              if (totalRaised < GOAL) {
50                  state = State.UNSUCCESSFUL;
51              } else {
52                  state = State.SUCCESSFUL;
53              }
54          }
55      }
56  }
```

Listing 6.2: Solidity Crowdfunding Contract

We also highlight the power and flexibility of querying which Chainlog offers. In defining all the information as logic rules, we have, without additional programming effort, made any of these predicates able to be queried through the Chainlog Platform. Users can ask the contract questions like "is the state successful?" through the blockchain's `queryContract` query with the goal `state(successful)`, or "what is the state?" through a query with the goal `state(State)`. They can also compose much more complex queries like "who are the funders who contributed more than 10000?" through the query `funded(Funder, Contribution), Contribution > 10000` (perhaps these most distinguished donors are entitled to a special perk). Supporting this last query in the Solidity contract is much more cumbersome. It requires an additional function, shown in listing 6.3, to be written.

```
1       mapping(address => uint) public contributions;
2       address[] public funders;
3
4       function getFundersAboveX(uint x) external view returns (address[] memory) {
5           address[] memory fundersAboveX = new address[](funders.length);
6           uint nextIndex = 0;
7           for (uint i = 0; i < funders.length; i++) {
8               address funder = funders[i];
9               if (contributions[funder] > x) {
10                  fundersAboveX[nextIndex++] = funder;
11              }
12          }
13          address[] memory result = new address[](nextIndex);
14          for (uint i = 0; i < nextIndex; i++) {
15              result[i] = fundersAboveX[i];
16          }
17          return result;
```

```
18     }
```
Listing 6.3: Solidity Crowdfunding – Funders Above X

Unlike some languages, mappings in Solidity are not iterable, so one must add and maintain an array to track who the funders are. The function loops over this array. The programmer must contend with the restriction that in-memory arrays in Solidity cannot grow dynamically in size, so they must initially overallocate a temporary array and then copy the results into a new, correctly sized array once the number of matching funders becomes known. Note that since smart contract code is immutable once deployed, the programmer must anticipate the need for this function beforehand. It cannot be added later.

6.1.2 Licence

For our second example, we revisit the licensing smart contract from "Evaluation of Logic-Based Smart Contracts for Blockchain Systems" [5] and its follow up [39]. The terms are found in example 3.1.

Listing 6.4 gives the snippet of the Solidity code that evaluates the contractual terms. The code is taken from [39], but has been modified to add an explicit termination variable. We assume that an arbiter tells the contract when external events occur, such as the grant of licence or the commission to publish, by way of setter functions. These setter functions have been omitted here as they are straightforward and not pertinent to this discussion. For the complete code, refer to appendix A.2.2.

```
1      bool public hasLicense;
2      bool public hasApproval;
3      bool public isCommissioned;
4
5      bool public use;
6      bool public usePermission;
7      bool public useForbidden;
8
9      bool public publish;
10     bool public publishPermission;
11     bool public publishForbidden;
12     bool public publishObligation;
13
14     bool public comment;
15     bool public commentPermission;
16     bool public commentForbidden;
17
18     bool public remove;
19     bool public removeObligation;
20
21     bool public violation;
22     bool public terminated;
23
24     constructor() {
25         useForbidden = true;
26         publishForbidden = true;
27         commentForbidden = true;
28     }
29
30     function evaluateLicenseContract() public {
31         // Article 1
32         if (hasLicense) {
33             useForbidden = false;
34             usePermission = true;
35         }
36         // Articles 2 and 4
37         if (hasLicense && (hasApproval || isCommissioned)) {
38             publishForbidden = false;
39             publishPermission = true;
40         }
41         // Article 2
42         if (hasLicense && !hasApproval && !isCommissioned && publish) {
43             removeObligation = true;
```

```
        }
        // Article 3
        if (publishPermission) {
            commentForbidden = false;
            commentPermission = true;
        }
        // Article 4
        if (hasLicense && isCommissioned) {
            publishForbidden = false;
            publishPermission = true;
            publishObligation = true;
        }
        // Article 5
        if (useForbidden && use
            || publishForbidden && publish
            || publishObligation && !publish
            || commentForbidden && comment
            || removeObligation && !remove) {
            violation = true;
            terminated = true;
        }
    }
```

Listing 6.4: Solidity Licence Contract

Procedural languages like Solidity require the programmer to manually articulate the changes to be made to the permissions, obligations and prohibitions. The burden rests on them to ensure that the program's instructions correctly encode the reasoning of the contract. In this case for example, when an obligation for an action is set to true, the programmer must see to it that the permission is also set to true and the prohibition is set to false. As a result, the process of translating a contract into Solidity is prone to errors.

The Chainlog equivalent is given in listing 6.5. In the logic programming paradigm, the programmer specifies what the rules of the contract are, but does not need to outline step by step how to accomplish them. The high-level, logic-based rules can be seen as an executable specification, which implies the program is guaranteed to be correct with respect to this specification. For example, when the programmer writes the rule `permission(A) :- action(A), obligation(A)`, which can be read as "if the licensee has the obligation to perform an action A, then they have the permission to perform A", they obtain the guarantee that if the obligation is true, so too will the permission be. Thus, errors to do with bookkeeping of variables are eliminated entirely.

```
action(A) :- member(A, [use, publish, comment, remove]).

forbidden(A) :- action(A), not permission(A).
permission(A) :- action(A), obligation(A).

% Article 1
permission(use) :- has_license.

% Article 2
permission(publish) :- has_license, has_approval.
obligation(remove) :- forbidden(publish), publish.

% Article 3
permission(comment) :- permission(publish).

% Article 4
obligation(publish) :- has_license, is_commissioned.

% Article 5
violation :- action(A), forbidden(A), A.
violation :- action(A), obligation(A), not A.
terminated :- violation.
```

Listing 6.5: Chainlog Licence Contract

Finally, we contrast the Chainlog code to the FCL expression from [5] shown in listing 3.1. As Governatori et al. did not provide an implementation, this is of course not actually executable as a smart contract; nevertheless it is still instructive to draw comparison with their vision. FCL is

a deontic logic – its rules are designed to be evaluated by a defeasible logic engine. Hence, the normative statuses of obligation, permission and prohibition are built-in.

Chainlog is not inherently deontic. However, its support for metaprogramming and negation as failure give it the ability to express deontic notions to some extent. The nature of this example is such that the licensee begins with all actions prohibited and is incrementally granted permissions and obligations. In FCL, this is captured by the superiority relation: article 1.0 dictates the prohibition for use, but when `hasLicense` becomes true, article 1.1 overrides article 1.0 and the permission is granted. In Chainlog on the other hand, we are able to use negation as failure to define prohibition "by default" in the absence of evidence for the corresponding permission (line 3). This approach is much more flexible, as it means Chainlog is not constricted to the highly specialised deontic logic and can express a much wider range of contracts.

6.1.3 Parametric Insurance

In contrast to indemnity insurance, which offers payouts based on the degree of damage or loss, parametric insurance compensates the policyholder when certain pre-defined parameters are met [58]. The parameters are based on measurable data. For example, a flight delay insurance policy might pay out when a flight is delayed for more than two hours.

Blockchain-based smart contracts lend themselves well to parametric insurance. When a claim is made, a smart contract is able to check the policy parameters programmatically. The process is fast and inexpensive, bypassing the need for a claims investigation. As the smart contract resides on a public, immutable ledger, the terms of the policy are made transparent to the policyholder and cannot be altered [58].

The example that follows is of a parametric disaster insurance contract between an insurer and a policyholder. Both the policy and the smart contracts have been created as part of this work, but the terms take inspiration from real-life parametric insurance products [59, 60]. The policy covers typhoons (tropical cyclones), earthquakes and flooding, in a region of Japan prone to these disasters. Payouts are made based on reliable data from the Japan Meteorological Agency and nearby weather stations: when earthquake seismic intensity, wind speed or rainfall measurements exceed agreed-upon thresholds, the policyholder becomes eligible to make a claim. Described below are the details:

Tropical Cyclone Cover

The policyholder is eligible for a payout under the tropical cyclone cover, provided such a claim has not been made for at least 4 weeks prior, in accordance with the following:

- 100% payout in the event of a tropical cyclone classified as 'violent typhoon'.
- 80% payout in the event of a tropical cyclone classified as 'very strong typhoon'.
- 40% payout in the event of a tropical cyclone classified as 'typhoon'.

Cyclone classification follows the Japan Meteorological Agency (JMA) Tropical Cyclone Intensity Scale, itself based on maximum 10-minute sustained wind speed. See table 6.1.

Category	Wind Speed Range (Knots)
Tropical Depression	Speed $<$ 34kt
Tropical Storm	34kt $<=$ Speed $<$ 48kt
Severe Tropical Storm	48kt $<=$ Speed $<$ 64kt
Typhoon	64kt $<=$ Speed $<$ 85kt
Very Strong Typhoon	85kt $<=$ Speed $<$ 105kt
Violent Typhoon	Speed $>=$ 105kt

Table 6.1: JMA Tropical Cyclone Classification [61]

Coverage limit: 100 units of cryptocurrency.

Earthquake Cover

The policyholder is eligible for a payout under the earthquake cover, provided such a claim has not been made for at least 4 weeks prior, in accordance with the following:

- 100% payout in the event of an intensity 6+ or 7 earthquake.
- 80% payout in the event of an intensity 6- earthquake.
- 40% payout in the event of an intensity 5+ earthquake.

Earthquake intensity follows the JMA Seismic Intensity Scale (Shindo Scale).

Coverage limit: 100 units of cryptocurrency.

Flood Cover

The policyholder is eligible for a payout under the flood cover, provided such a claim has not been made for at least 4 weeks prior, in accordance with the following:

- 100% payout in the event of rainfall exceeding 800mm over a ten-day period.

Coverage limit: 10 units of cryptocurrency.

As this example is relatively complicated, it is worth explaining the lifecycle of the smart contract:

- The contract is formed between insurer and policyholder and the code is written. Included are the insurer, data source and policyholder's account addresses, the policy expiration date and policy parameters such as the payout limits.
- The insurer initiates the contract by uploading it to the Chainlog Platform in a `createContract` transaction. This transaction will also include a sum of money that will be held in the contract for payouts.
- Wind speed, seismic intensity and rainfall data are continually input into the smart contract. The insurance company will have infrastructure in place for this to be uploaded automatically. The smart contract receives the data via `inputDisasterData` message calls from the data source address. One datapoint for each metric is added per day.
- If a disaster occurs and the policyholder becomes eligible to claim, they send a `claimPayout` message to the contract. The contract calculates the payout to be made and transfers the money to the policyholder's account.
- After the expiration date, the insurer terminates the contract by invoking `terminate`. The contract transfers all of its remaining funds back to the insurer.

Given this, listings 6.6 and 6.7 present the smart contract as expressed in Chainlog and Solidity respectively. Note that in Solidity, the `inputDisasterData` is divided into three functions with one for each of the three different types of data – this is because the datatypes are different (e.g. `uint` for wind speed, `string` for seismic intensity). `claimPayout` is also split as the bookkeeping is different for different policies.

```
1  policyholder('chainlog103v2kw0xnhdfrnzukphheq9zlm67xg8ykjhfrc').
2  data_source('chainlog1rdhj3kcg9kr9y7pm489z579n9eqcsqyq0rjmwd').
3  insurer('chainlog12t7x9c08wyuurnw7ayhjvv7ycev29tstudcxrp').
4
5  policy_expiration(1672534800).
6
7  payout_limit(cyclone, 100).
8  payout_limit(earthquake, 100).
9  payout_limit(flood, 10).
10
11 % Typhoon policy
12 cyclone_category(violent_typhoon, T)  :- wind_speed(Speed, T), Speed >= 105.
13 cyclone_category(v_strong_typhoon, T) :- wind_speed(Speed, T), Speed >= 85, Speed < 105.
14 cyclone_category(typhoon, T)          :- wind_speed(Speed, T), Speed >= 64, Speed < 85.
15
16 payout_percentage(cyclone, 100, T) :- cyclone_category(violent_typhoon, T).
17 payout_percentage(cyclone, 80, T)  :- cyclone_category(v_strong_typhoon, T).
```

```
payout_percentage(cyclone, 40, T)  :- cyclone_category(typhoon, T).

% Earthquake policy
payout_percentage(earthquake, 100, T) :- seismic_intensity(Intensity, T),
                                          (Intensity = '6+' or
                                           Intensity = '7').
payout_percentage(earthquake, 80, T)  :- seismic_intensity(Intensity, T), Intensity = '6-'.
payout_percentage(earthquake, 40, T)  :- seismic_intensity(Intensity, T), Intensity = '5+'.

% Flood policy
ten_day_rainfall(TenDayMM, T) :-
  rainfall(_, T),
  add_time(T, days(10), T2),
  sum_all(OneDayMM, (rainfall(OneDayMM, Day), Day >= T, Day < T2), TenDayMM).

payout_percentage(flood, 100, T) :- ten_day_rainfall(TenDayMM, T), TenDayMM > 800.

% Eligibility to claim payout
eligible_for(Disaster, PercentagePayout) :-
  payout_percentage(Disaster, PercentagePayout, T),
  not (claimed(Disaster, ClaimT),
       add_time(ClaimT, weeks(4), T1),
       T < T1).

% Message handlers
on inputDisasterData(Data):
  require sender(Sender), data_source(Sender);
  do      assert(Data).

on claimPayout(Disaster):
  require sender(Sender), policyholder(Sender);
  require time(Now), policy_expiration(ExpT), Now < ExpT;
  require eligible_for(Disaster, PercentagePayout);

  if      payout_limit(Disaster, Limit),
          Payout is Limit * PercentagePayout / 100;

  do      assert(claimed(Disaster, Now)),
          transfer(Sender, Payout).

on terminate:
  require time(T), policy_expiration(ExpirationDate), T > ExpirationDate;
  if      insurer(Insurer), balance(Balance);
  do      transfer(Insurer, Balance).
```

Listing 6.6: Chainlog Parametric Insurance Contract

```
pragma solidity ^0.8;

contract Insurance {

    address public constant POLICYHOLDER = 0x4A02caF9c411332F5917af19289FDC74A6fB079c;
    address public constant DATA_SOURCE = 0xF237A1826aA6Fc931B04bDeE1536224BF7E63C90;
    address public constant INSURER = 0x0460c3fb9ce497308777020199Eb553F31E1bcE7;

    uint public constant POLICY_EXPIRATION = 1672534800;

    uint public constant CYCLONE_PAYOUT_LIMIT = 100 ether;
    uint public constant EARTHQUAKE_PAYOUT_LIMIT = 100 ether;
    uint public constant FLOOD_PAYOUT_LIMIT = 10 ether;

    uint public cyclonePayoutPercentage = 0;
    uint public earthquakePayoutPercentage = 0;
    uint public floodPayoutPercentage = 0;

    uint public nextCycloneEligibleTime;
    uint public nextEarthquakeEligibleTime;
    uint public nextFloodEligibleTime;

    uint internal constant VIOLENT_TYPHOON_WIND_SPEED = 105;
    uint internal constant V_STRONG_TYPHOON_WIND_SPEED = 85;
    uint internal constant TYPHOON_WIND_SPEED = 64;

    uint[10] internal rainfallRecordings;
```

```
28      uint internal nextRainfallIndex = 0;
29      uint internal rainfall10DaySum = 0;
30
31      constructor() payable {}
32
33      modifier onlyDataSource() {
34          require(msg.sender == DATA_SOURCE, "caller is not the data source");
35          _;
36      }
37
38      modifier onlyPolicyholder() {
39          require(msg.sender == POLICYHOLDER, "caller is not the policyholder");
40          _;
41      }
42
43      modifier beforeExpiration() {
44          require(block.timestamp < POLICY_EXPIRATION, "policy has expired");
45          _;
46      }
47
48      function inputWindSpeedData(uint speed, uint timestamp) external onlyDataSource {
49          // Ignore data if not eligible for cyclone payout yet.
50          if (timestamp < nextCycloneEligibleTime) {
51              return;
52          }
53
54          // Compare speed against policy parameters.
55          if (speed >= VIOLENT_TYPHOON_WIND_SPEED) {
56              cyclonePayoutPercentage = 100;
57          } else if (speed >= V_STRONG_TYPHOON_WIND_SPEED) {
58              if (cyclonePayoutPercentage < 80) {
59                  cyclonePayoutPercentage = 80;
60              }
61          } else if (speed >= TYPHOON_WIND_SPEED) {
62              if (cyclonePayoutPercentage < 40) {
63                  cyclonePayoutPercentage = 40;
64              }
65          }
66      }
67
68      function inputSeismicIntensityData(string calldata intensity, uint timestamp) external
         onlyDataSource {
69          // Ignore data if not eligible for earthquake payout yet.
70          if (timestamp < nextEarthquakeEligibleTime) {
71              return;
72          }
73
74          // Compare intensity against policy parameters.
75          if (stringsEqual(intensity, "6+") || stringsEqual(intensity, "7")) {
76              earthquakePayoutPercentage = 100;
77          } else if (stringsEqual(intensity, "6-")) {
78              if (earthquakePayoutPercentage < 80) {
79                  earthquakePayoutPercentage = 80;
80              }
81          } else if (stringsEqual(intensity, "5+")) {
82              if (earthquakePayoutPercentage < 40) {
83                  earthquakePayoutPercentage = 40;
84              }
85          }
86      }
87
88      function inputRainfallData(uint rainfall, uint timestamp) external onlyDataSource {
89          // Ignore data if not eligible for flood payout yet.
90          if (timestamp < nextFloodEligibleTime) {
91              return;
92          }
93
94          // Update the circular queue and running 10 day sum.
95          rainfall10DaySum = rainfall10DaySum + rainfall - rainfallRecordings[nextRainfallIndex];
96          rainfallRecordings[nextRainfallIndex] = rainfall;
97          nextRainfallIndex = (nextRainfallIndex + 1) % 10;
98
99          // Compare 10 day sum against policy parameter.
```

```
100          if (rainfall10DaySum > 800) {
101              floodPayoutPercentage = 100;
102          }
103      }
104
105      function claimCyclonePayout() external onlyPolicyholder beforeExpiration {
106          require(cyclonePayoutPercentage > 0, "not eligible for cyclone payout");
107          uint payout = CYCLONE_PAYOUT_LIMIT * cyclonePayoutPercentage / 100;
108
109          nextCycloneEligibleTime = block.timestamp + 4 weeks;
110          cyclonePayoutPercentage = 0;
111
112          payable(msg.sender).transfer(payout);
113      }
114
115      function claimEarthquakePayout() external onlyPolicyholder beforeExpiration {
116          require(earthquakePayoutPercentage > 0, "not eligible for earthquake payout");
117          uint payout = EARTHQUAKE_PAYOUT_LIMIT * earthquakePayoutPercentage / 100;
118
119          nextEarthquakeEligibleTime = block.timestamp + 4 weeks;
120          earthquakePayoutPercentage = 0;
121
122          payable(msg.sender).transfer(payout);
123      }
124
125      function claimFloodPayout() external onlyPolicyholder beforeExpiration {
126          require(floodPayoutPercentage > 0, "not eligible for flood payout");
127
128          nextFloodEligibleTime = block.timestamp + 4 weeks;
129          floodPayoutPercentage = 0;
130          delete rainfallRecordings;
131          rainfall10DaySum = 0;
132
133          payable(msg.sender).transfer(FLOOD_PAYOUT_LIMIT);
134      }
135
136      function terminate() external {
137          require(msg.sender == INSURER, "caller is not the insurer");
138          require(block.timestamp > POLICY_EXPIRATION, "policy has not expired yet");
139
140          payable(msg.sender).transfer(address(this).balance);
141      }
142
143      function stringsEqual(string calldata s1, string memory s2) internal pure returns (bool) {
144          return keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2));
145      }
146  }
```

Listing 6.7: Solidity Parametric Insurance Contract

Both contracts begin by defining constants such as the stakeholders' addresses, expiration timestamp and payout limits. In Solidity, this is done via public constants and state variables. When a state variable is declared public, the Solidity compiler will automatically generate a getter method so that its value can be queried. Chainlog, on the other hand, uses facts as part of the static KB. The nature of Chainlog's logic queries means that it is not necessary to generate any additional "getter" code; one is able to submit a logic query such as `policy_expiration(T)` to discover the expiration date. Alternatively, they can request the contract code to see all of the facts at once, since the program is stored as readable source code as opposed to inscrutable EVM bytecode.

The policy parameters are evaluated very differently by the two languages. Consider first the tropical cyclone policy. In the imperative paradigm of Solidity, a state variable `cyclonePayoutPercentage` (line 15) is used to track the highest payout percentage the policyholder is entitled to based on the data encountered so far. When new wind speed data comes in via `inputWindSpeedData` (lines 48-66), an if-else block determines whether and how to update this variable.

In Chainlog, data is simply added to the dynamic KB by the `assert` on line 45. Wind speed data takes the form `wind_speed(Speed, Timestamp)`. The payout percentage is then defined intensionally by logical rules on the KB (lines 12-18). This representation is arguably more readable and closer to the natural language policy description given earlier. Consider the rule on line 16 for example:

this can be read as "the cyclone payout percentage becomes 100% at time T if there was a violent typhoon at T", which very closely mimics the first item under the cyclone cover.

The earthquake policy is similar. In Solidity, a state variable `earthquakePayoutPercentage` (line 16) tracks the highest payout percentage so far. In this case, intensity is given as a string in order to allow for intensities like "5+". Solidity strings cannot be compared for equality directly, so a helper function `stringsEqual` is introduced on line 143 to perform the check by comparing the raw bytes of their Keccak-256 hashes. As strings are a reference type, the programmer is now burdened with specifying the data location of the strings: in this case, the first string will originally come from the function parameter to `inputSeismicIntensityData` and therefore resides in the `calldata`, while the second will be a string literal with the appropriate location being `memory`.

The Chainlog programmer is relieved from such arcane concerns as the need to deal with raw string bytes and the need to consider the location in which data resides. The earthquake policy is expressed in three concise, declarative rules on lines 21-25. Chainlog allows the programmer to program at a higher level, where one specifies what the policy rules are, rather than how to achieve them.

The flood policy is the most involved. Here, the payout trigger is determined by the ten-day rainfall; however, recall that the data source uploads one-day rainfall measurements to the contract. It no longer suffices in Solidity to store a single state variable, as we require the sum of the last ten measurements. One way to keep track of this is to use an array of size ten as a circular queue, illustrated in figure 6.1. The running ten-day sum and an index into the queue are maintained. As new data is input, we overwrite the value under the current index, simultaneously enqueuing the new measurement and dequeuing the oldest measurement. While doing so, we subtract the dequeued value from the running ten-day sum and add to it the enqueued value. The index is advanced, wrapping to the front of the array when overrunning the end. The ten-day sum is compared against the policy parameter of 800mm. Lines 95-102 implement this algorithm.

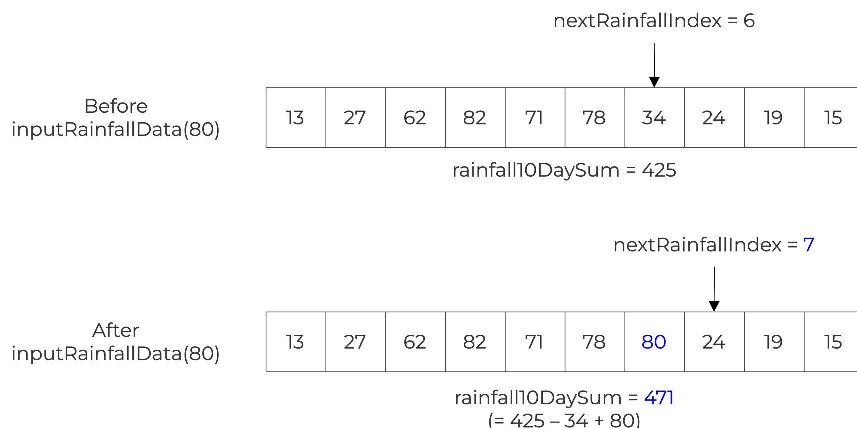

Figure 6.1: Ten-day Rainfall Circular Queue

The Chainlog code for the flood policy is seen in lines 28-33. `ten_day_rainfall/2` defines the ten-day rainfall from a given time `T` declaratively: given there was rainfall at `T`, and `T2` is 10 days after `T`, the ten-day rainfall is the sum of all one-day rainfalls between `T` and `T2`. The programmer does not need to be concerned with maintaining a circular queue data structure. Chainlog is high-level – the code defines what the ten-day rainfall is, rather than the steps to take in order to compute it.

6.1.4 Pharmacogenomics Data

This example concerns the storage and querying of gene-drug relationship data, a branch of pharmaceutical data which describes the effect of genetic variation on drug efficacy and adverse drug reactions. By consulting the data for a given patient's genetic variants, clinicians and researchers can predict how the patient will respond when administered a given drug [62].

Data integrity is of crucial importance in pharmacogenomics [63]. Blockchain technology addresses many of the challenges with existing centralised storage methods: notably, decentralisation avoids data loss due to a single-point-of-failure, and immutability prevents alteration and corruption of records [62].

The smart contract of this section manages a database of empirical observations. Each record is a six-tuple $\langle Gene, Variant, Drug, Outcome, Relation, SideEffect \rangle$ corresponding to one specific patient observation. Table 6.2 provides an overview of the fields.

Field	Description	Example
Gene	Gene name	HLA-B
Variant	Variant number	57
Drug	Drug name	abacavir
Outcome	Outcome – one of Improved, Unchanged, Deteriorated	Improved
Relation	Whether a gene outcome relation is suspected	true
Side Effect	Whether serious side effects were observed	false

Table 6.2: Gene-drug Interaction Fields. Adapted from [62]

The smart contract will allow the insertion of single observations. It will support querying of the data by any combination of gene name, variant number and drug name. For example, a query may request for all observations with a gene name of "HLA-B" and variant number of 57.

The Solidity contract originates from Gürsoy et al. [63] and related work [64, 65]. Listing 6.8 presents a section from a simplified version.

```
1  pragma solidity ^0.8;
2
3  contract PharmacogenomicsData {
4
5      struct Observation {
6          string gene;
7          uint variant;
8          string drug;
9          string outcome;
10         bool relation;
11         bool sideEffect;
12     }
13
14     mapping(uint => Observation) public database;
15     uint public counter;
16
17     mapping(string => uint[]) public geneMapping;
18     mapping(uint => uint[]) public variantMapping;
19     mapping(string => uint[]) public drugMapping;
20
21     function insert(
22         string calldata gene,
23         uint variant,
24         string calldata drug,
25         string calldata outcome,
26         bool relation,
27         bool sideEffect
28     ) external {
29         geneMapping[gene].push(counter);
30         variantMapping[variant].push(counter);
31         drugMapping[drug].push(counter);
32         database[counter] = Observation(gene, variant, drug, outcome, relation, sideEffect);
33         counter++;
34     }
35
36     function query(
37         string calldata gene,
38         string calldata variant,
39         string calldata drug
40     ) external view returns (Observation[] memory) {
41         uint[] memory idList = new uint[](counter);
42         uint matchCount = 0;
43         uint[] memory genes;
```

```
44         uint[] memory variants;
45         uint[] memory drugs;
46 
47         // If database empty, return empty
48         if (counter == 0) {
49             return new Observation[](0);
50         }
51 
52         // Check number of fields being searched by
53         uint len = 0;
54         if (!stringsEqual(gene, "*")) {
55             len++;
56             genes = geneMapping[gene];
57             if (genes.length == 0) {
58                 return new Observation[](0);
59             }
60         }
61         if (!stringsEqual(variant, "*")) {
62             len++;
63             (bool success, uint variantUint) = stringToUint(variant);
64             require(success, "variant is not a valid uint");
65             variants = variantMapping[variantUint];
66             if (variants.length == 0) {
67                 return new Observation[](0);
68             }
69         }
70         if (!stringsEqual(drug, "*")) {
71             len++;
72             drugs = drugMapping[drug];
73             if (drugs.length == 0) {
74                 return new Observation[](0);
75             }
76         }
77 
78         if (len == 0) {
79             // All fields are wildcards; push all IDs in database
80             matchCount = counter;
81             for (uint i = 0; i < counter; i++) {
82                 idList[i] = i;
83             }
84         } else {
85             // Compute intersection of genes, variants and drugs
86             matchCount = computeIntersection(genes, variants, drugs, len, matchCount, idList);
87         }
88 
89         Observation[] memory results = new Observation[](matchCount);
90         for (uint i = 0; i < matchCount; i++) {
91             results[i] = database[idList[i]];
92         }
93         return results;
94     }
95 
96     function stringsEqual(
97         string calldata s1,
98         string memory s2
99     ) internal pure returns (bool) {
100        return keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2));
101    }
102
103    function stringToUint(
104        string calldata s
105    ) internal pure returns (bool success, uint result) {
106        bytes memory b = bytes(s);
107        uint j = 1;
108        for (uint i = b.length - 1; i >= 0 && i < b.length;) {
109            if (!(uint8(b[i]) >= 48 && uint8(b[i]) <= 57)) {
110                return (false, 0);
111            }
112            result += j * (uint8(b[i]) - 48);
113            j *= 10;
114            unchecked {
115                i--;
116            }
```

```
117        }
118        success = true;
119    }
```

Listing 6.8: Solidity Pharmacogenomics Data Contract

The database is a mapping of unique IDs to observation structs declared on line 14. To facilitate search, three indices for the three query fields (genes, variants and drugs) are maintained. The gene index, declared on line 17, is a mapping from a gene string to the array of IDs of the observations with that gene. Likewise for variants and drugs on lines 17 and 18.

The query function defined on lines 36-94 accepts strings for the gene, variant and drug to search by. Any combination of these parameters can be the special wildcard string `"*"` to avoid restricting results by that field. For example, the aforementioned query for gene name "HLA-B" and variant number 57 would translate to a call `query("HLA-B", "57", "*")`; notice the wildcard in the drug field. The query algorithm by Gürsoy et al. works as follows: if the database is empty, return empty immediately. Otherwise, an array `idList` of IDs of observations that match the query will be populated. To do this, the code first determines the number of fields being searched by – i.e. the number of non-wildcard fields. If this is zero, `idList` includes all IDs in the entire database. Otherwise, it is the intersection of the arrays found by looking up the relevant gene, variant and drug indices. The intersection algorithm is omitted here as the method used by Gürsoy et al. is very long; the reader is referred to appendix A.4.2 for the full code. Finally, the array of observations to be returned is constructed by looking up all the IDs in `idList` in the database mapping and fetching their `Observation` structs.

One quirk of the query method is that the variant is received as a string and not an integer. This is necessary to support the wildcard string. However, it creates complications when attempting to look up the variants mapping, which uses an integer key. Solidity provides no native conversion from numeric string to integer, so this also demands the programmer write the helper function on lines 103-119 to loop over the bytes of the string and multiply each digit by the appropriate power of ten. Writing this function can be the source of several subtle bugs, even for the experienced programmer.

For one, the loop cannot be written simply as `for (uint i = b.length; i >= 0; i--)`, as after the final iteration of the loop, the execution of `i--` will attempt to subtract 1 from 0, underflowing the range of the unsigned 256-bit integer type `uint` (the type of `i` must be of unsigned integer as it is used as an array index). As of Solidity version 0.8.0, integer underflow causes a `REVERT` operation, meaning this code would cause a transaction to fail when attempting to query the database with a non-wildcard variant. The programmer must be aware of this update to the language and explicitly place the `i--` inside an `unchecked` block to prevent the revert.

Even after introducing unchecked arithmetic, the programmer must now be aware that upon unchecked integer underflow, the value wraps around to the highest possible `uint`. Thus, after subtracting 1 from 0, the loop terminating condition of `i >= 0` remains true, and the loop does not stop. This will lead to an error when attempting to index the byte array, as the value of `i` will be out of bounds. The programmer must add `i < b.length` to the loop condition to avoid this error.

We highlight the fact that these bugs only manifest at runtime. The Solidity compiler will happily accept the buggy code, and it is conceivable for a developer to deploy a buggy contract to the Ethereum mainnet without ever being aware of the errors. One must have prior knowledge of the relevant language behaviour or must test the contract thoroughly to uncover these bugs. Indeed, vulnerabilities to do with overflow/underflow in Solidity have led to disastrous consequences in the past [66].

```
1 query(Gene, Variant, Drug, Observations) :-
2   find_all(observation(Gene, Variant, Drug, Outcome, Relation, SideEffect),
3            observation(Gene, Variant, Drug, Outcome, Relation, SideEffect),
4            Observations).
5
6 on insert(Gene, Variant, Drug, Outcome, Relation, SideEffect):
7   do assert(observation(Gene, Variant, Drug, Outcome, Relation, SideEffect)).
```

Listing 6.9: Chainlog Pharmacogenomics Data Contract

Let us now turn to the Chainlog implementation, shown in listing 6.9. Immediately obvious is the fact that it is much more compact – only 7 lines compared to 216. The inclusion of the second-order predicate `find_all/3` in the language means all of the querying work can be done by Chainlog's backward search. The programmer does not have to be concerned with the data structures (structs, mappings, arrays) used to store the data, nor do they have to explicitly detail the steps needed to evaluate the query as they do in Solidity.

Existential quantification of variables in logic queries means that no additional effort is required to support wildcards: to search for gene "HLA-B" and variant 57, one submits `query('HLA-B', 57, _, Obs)` with a variable in the drug field. The power and flexibility of queries inherently possible on Chainlog contracts far exceeds that of Ethereum contracts.

By abstracting away the details about how types are stored and compared, the bugs to do with string and integer conversion discussed earlier become impossible to make in Chainlog. When the Solidity contract requires that the programmer write their own algorithms for finding the intersection between arrays, etc., it opens up opportunities for making logical errors. A high-level language like Chainlog allows the programmer to state what they want without needing to specify how to compute it; they instead defer to the underlying inference mechanisms. This leads to more bug-free programs.

6.2 Security

Security is a critical concern in the blockchain context. Throughout this work, we have committed to a host of design and implementation decisions to protect against potential attacks and ensure the secure operation of the Chainlog blockchain. We highlight the outcome of these measures in this section.

6.2.1 Denial of Service Protection

A blockchain system must be robust to denial of service (DoS) attacks. Any opportunity for an attacker to create a disproportionate amount of work for the blockchain's nodes with relatively little effort or cost is potentially a vector for this type of attack, as it would occupy the nodes from being able to process new transactions.

The primary defence employed by the Chainlog Platform is the imposition of the depth limit in SLDNF*. This places a finite limit on the amount of computation a query can spawn, and ensures execution always terminates.

The depth limit guards against programs that are deep; however, it does not prevent programs from being "wide" – i.e., containing a huge number of clauses. To protect against this, we also impose a byte size limit on smart contract code. If an attacker attempts to upload an unmanageably large smart contract, the transaction will be rejected during the `CheckTx` validation phase, and will not be broadcast to the network.

Although not covered in the implementation chapter, the Chainlog Platform does include a rudimentary gas system through utilities provided by the Cosmos SDK. Each transaction incurs an initial gas cost proportional to its size. The deduction of this fee is processed by the AnteHandler; if the sender cannot pay the fee, the AnteHandler instructs Tendermint to discard the transaction at the `CheckTx` stage. This protects the Chainlog Platform from spam: it would be infeasible cost-wise for an attacker to flood the blockchain network with garbage transactions.

In addition, the key-value stores of the MultiStore use a wrapper which consumes gas for each read and write made to them. This prevents the possibility for an attacker to bloat the store with an unbounded number of facts added by `assert` actions, as each update to the dynamic KB would levy a fee.

6.2.2 Determinism

The secure and tamper-free properties of the blockchain rely on the fact that a transaction is verified and agreed upon by all nodes in the network. Hence, it is crucial for the state transition function to be fully deterministic. We have taken measures in both the language and blockchain design to ensure this.

SLDNF resolution is a non-deterministic search strategy. We created a modification, SLDNF*, which is deterministic. To achieve this, the logic clauses of Chainlog knowledge bases are given an order. A deterministic selection function is employed, choosing goal literals left-to-right and program clauses top-to-bottom. Furthermore, the order in which the knowledge bases are consulted follows a defined order. These reasons make Chainlog's query interpreter deterministic.

The message handlers of a Chainlog program are also ordered, and are searched by the message interpreter top-to-bottom. There is no ambiguity when multiple message handlers match a message term; the topmost one is always chosen. In addition, the unifications made during `if` and `require` clauses are deterministic because the query interpreter is deterministic. The execution of actions follows the order they are written, and `assert` actions always append to the end of the dynamic KB. For these reasons, the Chainlog message interpreter is also deterministic.

The Chainlog language has restrictions in place to eliminate all sources of non-determinism. Constructs for the access of files, streams, network and other processes have been excluded. Chainlog programs also do not have the ability to exit the process or make system calls.

Transactions on the Chainlog Platform may contain multiple messages. If this is the case, the platform ensures they are executed in the exact order they are given. During a `createContract` message, the new smart contract's address is not randomly generated; rather, it is produced by a deterministic transformation of the creator's address and sequence using a cryptographic hash function. Execution of contracts during a `callContract` message is completely isolated due to the aforementioned language restrictions. Because the local time on nodes' machines is likely to differ, smart contracts instead use the deterministic block timestamp as a source of time. As a result of these measures, the execution of transactions on the Chainlog Platform – and therefore its state transition function – obeys determinism.

6.2.3 Code Vulnerability Mitigation

Solidity code is susceptible to a wide range of vulnerabilities [23, 67, 68]. While many of them are not applicable to Chainlog because they involve Solidity-specific features, several of them are mitigated in our language. We discuss some of these here.

Integer overflow/underflow bugs – where unexpected behaviour arises due to integers exceeding their range – are eliminated in Chainlog. This is because Chainlog inherits its integer representation from Prolog, which supports numbers greater than 64 bits by chaining words [44].

Unprotected suicide concerns Solidity's `selfdestruct(address)` function, which deletes the contract's account (including its code) and sends all its remaining balance to `address`. This vulnerability is characterised by improper authorisation to the `selfdestruct` call [68]. By design, Chainlog does not support the deletion of contracts, as it opens the possibility for a party to evade the consequences of a contract when circumstances become unfavourable. Therefore, this vulnerability is simply not possible in Chainlog.

Secrecy failure occurs when confidential data is stored on the blockchain in unencrypted format. The vulnerability arises due to the misconception that Solidity variables with `private` visibility are hidden. `private` variables only prevent access from other contracts; blockchain data is publicly accessible to all. Chainlog does not support visibility modifiers and follows the simple rule that all predicates are publicly queryable. Hence, this misconception is mitigated.

Reentrancy attacks involve cyclic calls between smart contracts and are therefore not applicable to Chainlog in its current state. However, we remarked in the design extensions section that the message interpreter could be made to reorder the execution of actions, should there be support

for inter-contract calls in the future. By ordering contract calls after all `assert` and `retract` actions for example, Chainlog could prevent programmers from inadvertently writing code vulnerable to reentrancy.

6.2.4 User-level Concerns

The design and implementation of Chainlog provides a number of security and trust guarantees consequential to the platform's end users and stakeholders.

For a start, the Chainlog Platform performs all smart contract execution and inference on-chain. Contract execution is performed by and agreed between all nodes in the blockchain's network. It is therefore decentralised. There is no need for a trusted third-party inference server as in the off-chain solutions suggested in [5].

Chainlog's blockchain uses public key cryptography and an Elliptic Curve Digital Signature Algorithm (ECDSA) as a means of secure authentication. The secp256k1 scheme used in the Chainlog Platform is the same as that used in Bitcoin and many other blockchains [69], and has therefore been battle-tested. This prevents an attacker from impersonating another user.

Our system makes all code publicly accessible through the `contractCode` query. In contrast to Ethereum's bytecode contracts, Chainlog smart contracts are stored in the platform's state as human-readable source code, and do not require decompilation to be examined. For these reasons, Chainlog code is transparent and inspectable [70]. Users are able to scrutinise the terms encoded in any smart contract and verify that there is no malicious fine print.

Furthermore, the source code of a Chainlog smart contract is immutable once deployed. It is impossible for a party to alter the terms of a contract for personal benefit.

6.2.5 Other Security Considerations

The implementation of the Chainlog Platform incorporates a number of miscellaneous security practices. At the consensus level, the mechanism utilised is Byzantine Fault Tolerant, able to remain safe provided less than $\frac{1}{3}$ of validators are Byzantine [15]. At the application level, replay attacks are prevented by the use of the sequence number, which assigns a unique identifier to each transaction.

For the developer wishing to extend the Chainlog implementation, the use of an object-capability security model guards against unsafe and malicious accesses to store. This is accomplished through the modules' keepers. Developers are encouraged to adopt the principle of least privilege when writing modules: if the need arises to call to functionality in a different module, they should define a keeper to expose only the minimal access permissions required for that module to complete its task.

6.3 Summary and Extensions

Evidently, Chainlog is a high-level, expressive language capable of articulating a broad range of contracts. It possesses a powerful logic-based component which reads like a natural language specification. Compared to Solidity, writing smart contracts in Chainlog leads to more comprehensible and bug-free code, and mitigates a variety of vulnerabilities and programmer errors. The language supports powerful querying capabilities, enabling parties to ask complex questions to contracts without the need for any additional code to be written.

The Chainlog Platform is a robust, secure blockchain system tolerant to node failure and resistant to a range of attack vectors. The execution of smart contracts as part of an on-chain, deterministic state transition function enables parties to engage in agreements regulated by a trustless, decentralised network.

Together, the two accomplishments of this work satisfy our objective to facilitate the representation, deployment and execution of logic-based smart contracts and realise the benefits of the declarative formalism.

Chainlog is not a perfect however, and the language and platform do have limitations. Most notable is the inability for Chainlog contracts to interact with each other. This becomes pertinent for large projects where it is necessary to organise code into separate modules for manageability.

Performance and scalability were not goals of this work – nevertheless, an insightful extension might investigate the capacity for the Chainlog Platform to scale to thousands of smart contracts. An experiment could be run involving a worldwide network of hundreds of nodes, simulating thousands of `createContract` and `callContract` transactions over a fixed time period. Throughput measurements in terms of transactions processed per second would give an indication of the congestion of the network, and can be compared against existing smart contract platforms like Ethereum.

Chapter 7

Conclusion

At the beginning of this work, we set out to empower developers with a framework for expressing and executing smart contracts in a logic-based language. The Chainlog language and platform succeed in achieving this goal.

Chainlog is a new hybrid-paradigm language which combines the high-level declarations of normal logic clauses with the procedural capabilities of message handlers. We demonstrated the expressiveness of the language through a range of real world use cases, showing it to be closer to natural language, more comprehensible and less error-prone than Solidity.

It is complemented by our specification and implementation of the Chainlog Platform, a fully functional blockchain supporting the deployment and execution of Chainlog smart contracts. We designed the platform with security in mind, taking preventative measures against potential attacks such as denial of service, impersonation and double-spending. In addition, we saw that Chainlog smart contracts are stored in a transparent, inspectable manner.

7.1 Future Work

The scope for further work on both the Chainlog language and platform is broad. We have indicated several possible extensions throughout the report already, which we go into more depth here.

Gas Mechanism: we briefly mentioned the Chainlog Platform's basic gas scheme in the security evaluation, which deducts a fee proportional to the size of the transaction and for each access to the store. However, there is no cost associated with the execution of queries and messages, where the bulk of computation occurs. Ideally, gas costs should correspond to the amount of computational work done. This extension would see a modification to the query metainterpreter to consume gas for each logical inference made and throw an exception when it becomes depleted. The second phase of the message interpreter would also deduct gas for each action executed and fail if the gas is insufficient.

Inter-contract Interaction: one of the main deficiencies of Chainlog when compared to Solidity is the inability for contracts to make calls to each other. There are two ways to interact with a Chainlog contract: queries and messages. For the former, we could consider allowing contracts to reference predicates from other contracts inside the bodies of its own logic clauses and `if` and `require` conditions. For the latter, we could add a new action which submits a message term to another contract's message handler set. Together, these would allow smart contracts to both ask questions of and send instructions to other contracts.

Formal Verification: the declarative nature of logic-based languages means they are well amenable to formal verification. We envisage the mapping Chainlog of programs to reasoning tools such as theorem provers. This would further assist developers in writing safe, correct smart contracts.

7.2 Ethical and Legal Discussion

As with any programming language, it is possible for one to write Chainlog programs with malevolent or criminal intent. Our view is that it is the onus of the smart contract developer to use the language and platform responsibly.

The environmental impact of mining is a common concern with regards to blockchain technology. The Chainlog Platform uses a Proof-of-Stake consensus mechanism, which replaces the computationally intensive mining process of Proof-of-Work with a staking system and random validator selection. This issue therefore does not apply to this work.

An interesting question is whether or not smart legal contracts are legally enforceable. Naturally, this depends on legal jurisdiction and may differ from region to region. In principle, smart contracts are no different from existing electronic contracts, and constitute legally binding agreements provided the requirements for the formation of a legal contract are satisfied (i.e. offer, acceptance, consideration, intention to create legal relations) [71]. The England and Wales Law Commission clarified this in their advice to the government in late 2021. There they concluded that the existing legal principles of English common law can apply to smart contracts without the need for statutory law reform [72].

We have identified no other ethical issues relating to this work.

7.3 Final Remarks

Logic programming languages have traditionally not been as well-adopted as their imperative counterparts, but they have proven to be effective in certain niche applications within AI, optimisation, databases and more. We hope that this work has convinced the reader at least to consider the inclusion of blockchain smart contracts in this list.

We have been somewhat critical of Solidity within this work; however, it is not our intention to adjudge it inferior to Chainlog. While we have demonstrated Chainlog to be more suitable for expressing rules, there remain many applications where Solidity would be the better choice. A good example is tokens, which in Solidity is typically accomplished by inheriting from a contract or interface adhering to the ERC20 standard [73]. The object-oriented features of Solidity would be more appropriate here.

Both Solidity and Chainlog have their respective strengths. Solidity's object-oriented features facilitate modularity and composability. On the other hand, Chainlog excels at representing contracts with involving knowledge, rules and reasoning. As is often the case in Computer Science, it is a matter of choosing the right tool for the job.

Bibliography

[1] Samya Dhaiouir and Saïd Assar. A systematic literature review of blockchain-enabled smart contracts: Platforms, languages, consensus, applications and choice criteria. In Fabiano Dalpiaz, Jelena Zdravkovic, and Pericles Loucopoulos, editors, *Research Challenges in Information Science*, pages 249–266, Cham, 2020. Springer International Publishing.

[2] Ethereum. Solidity. https://docs.soliditylang.org/en/latest/, 2016. [Accessed 8th November 2021].

[3] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.org/en/whitepaper/, 2014.

[4] Ángel Jesús Varela-Vaca and Antonia M. Reina Quintero. Smart contract languages: A multivocal mapping study. *ACM Comput. Surv.*, 54(1), jan 2021. ISSN 0360-0300. doi: 10.1145/3423166. URL https://doi.org/10.1145/3423166.

[5] Guido Governatori, Florian Idelberger, Régis Riveret, and Giovanni Sartor. Evaluation of logic-based smart contracts for blockchain systems. In Jose Julio Alferes, Leopoldo Bertossi, Guido Governatori, Paul Fodor, and Dumitru Roman, editors, *Rule Technologies. Research, Tools, and Applications*, pages 167–183, Cham, 2016. Springer International Publishing. ISBN 978-3-319-42019-6.

[6] Ethereum. Blocks. https://ethereum.org/en/developers/docs/blocks/, 2022. [Accessed 16th June 2022].

[7] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg. ISBN 978-3-540-48184-3.

[8] Ethereum. Consensus mechanisms. https://ethereum.org/en/developers/docs/consensus-mechanisms/, 2022. [Accessed 16th June 2022].

[9] Sinan Küfeoğlu and Mahmut Özkuran. Bitcoin mining: A global review of energy and power demand. *Energy Research & Social Science*, 58:101273, Dec 2019. ISSN 2214-6296. URL https://www.sciencedirect.com/science/article/pii/S2214629619305948.

[10] Cong T. Nguyen, Dinh Thai Hoang, Diep N. Nguyen, Dusit Niyato, Huynh Tuong Nguyen, and Eryk Dutkiewicz. Proof-of-stake consensus mechanisms for future blockchain networks: Fundamentals, applications and opportunities. *IEEE Access*, 7:85727–85745, 2019. doi: 10.1109/ACCESS.2019.2925010.

[11] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2019.12.019. URL https://www.sciencedirect.com/science/article/pii/S0167739X19316280.

[12] Nick Szabo. Smart contracts: Building blocks for digital markets. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html, 1996.

[13] Pratima Sharma, Rajni Jindal, and Malaya Dutta Borah. A review of smart contract-based platforms, applications, and challenges. *Cluster Computing*, Jan 2022. ISSN 1573-7543. doi: 10.1007/s10586-021-03491-1. URL https://doi.org/10.1007/s10586-021-03491-1.

[14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf, 2014.

[15] Tendermint Inc. What is tendermint. https://docs.tendermint.com/master/introduction/what-is-tendermint.html, 2018. [Accessed 25th January 2022].

[16] Tendermint Inc. Cosmos sdk. https://docs.cosmos.network/main/intro/overview.html, 2022. [Accessed 13th June 2022].

[17] Ziyan Wang, Xiangping Chen, Xiaocong Zhou, Yuan Huang, Zibin Zheng, and Jiajing Wu. An empirical study of solidity language features. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 698–707, 2021. doi: 10.1109/QRS-C55045.2021.00105.

[18] Benjamin Cassidy. Smart contract languages: A thorough comparison. *Preprint*, 10 2020. doi: 10.13140/RG.2.2.22479.92326.

[19] Quang-Thang Nguyen, Bao Son Do, Thi Tam Nguyen, and Ba-Lam Do. Gassaver: A tool for solidity smart contract optimization. In *Proceedings of the Fourth ACM International Symposium on Blockchain and Secure Critical Infrastructure*, BSCI '22, page 125–134, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391757. doi: 10.1145/3494106.3528683. URL https://doi.org/10.1145/3494106.3528683.

[20] Ethereum. Solidity by example. https://docs.soliditylang.org/en/latest/solidity-by-example.html, 2016. [Accessed 18th January 2022].

[21] Franziska Heintel. Solidity developer survey 2021 results. https://blog.soliditylang.org/2022/02/07/solidity-developer-survey-2021-results/, 2022. [Accessed 17th June 2022].

[22] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 47(10):2084–2106, 2021. doi: 10.1109/TSE.2019.2942301.

[23] Aicha Bouichou, Soufiane Mezroui, and Ahmed El Oualkadi. An overview of ethereum and solidity vulnerabilities. In *2020 International Symposium on Advanced Electrical and Communication Technologies (ISAECT)*, pages 1–7, 2020. doi: 10.1109/ISAECT50560.2020.9523638.

[24] ConsenSys. Mythril. https://github.com/ConsenSys/mythril, 2022. [Accessed 17th June 2022].

[25] crytic. Slither. https://github.com/crytic/slither, 2022. [Accessed 17th June 2022].

[26] trailofbits. Manticore. https://github.com/trailofbits/manticore, 2022. [Accessed 17th June 2022].

[27] enzymefinance. Oyente. https://github.com/enzymefinance/oyente, 2020. [Accessed 17th June 2022].

[28] Anna Vacca, Andrea Di Sorbo, Corrado A. Visaggio, and Gerardo Canfora. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. *Journal of Systems and Software*, 174:110891, 2021. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2020.110891. URL https://www.sciencedirect.com/science/article/pii/S0164121220302818.

[29] Bin Hu, Zongyang Zhang, Jianwei Liu, Yizhong Liu, Jiayuan Yin, Rongxing Lu, and Xiaodong Lin. A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. *Patterns*, 2(2):100179, 2021. ISSN 2666-3899. doi: https://doi.org/10.1016/j.patter.2020.100179. URL https://www.sciencedirect.com/science/article/pii/S2666389920302439.

[30] Vyper Team. Vyper. https://vyper.readthedocs.io/en/stable/, 2017. [Accessed 2nd December 2021].

[31] Mudabbir Kaleem, Anastasia Mavridou, and Aron Laszka. Vyper: A security comparison with solidity based on common vulnerabilities. In *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 107–111, 2020. doi: 10.1109/BRAINS49436.2020.9223278.

[32] Vyper Team. Vyper by example. https://vyper.readthedocs.io/en/stable/vyper-by-example.html, 2017. [Accessed 18th January 2022].

[33] LIGO. Introduction to ligo. https://ligolang.org/docs/intro/introduction, 2022. [Accessed 17th June 2022].

[34] L.M Goodman. Tezos — a self-amending crypto-ledger. https://tezos.com/whitepaper.pdf, 2014.

[35] Michael Coblenz. Obsidian: A safer blockchain programming language. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 97–99, 2017. doi: 10.1109/ICSE-C.2017.150.

[36] Hyperledger. Hyperledger fabric. https://hyperledger-fabric.readthedocs.io/en/latest/whatis.html, 2022. [Accessed 17th June 2022].

[37] pirapira. Bamboo: a language for morphing smart contracts. https://github.com/pirapira/bamboo, 2018. [Accessed 17th June 2022].

[38] Franklin Schrans, Daniel Hails, Alexander Harkness, Sophia Drossopoulou, and Susan Eisenbach. Flint for safer smart contracts, 2019. URL https://arxiv.org/abs/1904.06534.

[39] Guido Governatori, Florian Idelberger, Zoran Milosevic, Regis Riveret, Giovanni Sartor, and Xiwei Xu. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law*, 26(4):377–409, Dec 2018. ISSN 1572-8382. doi: 10.1007/s10506-018-9223-3. URL https://doi.org/10.1007/s10506-018-9223-3.

[40] Alfredo Maffi. Blockchain and beyond: Proactive logic smart contracts. Master's thesis, Università di Bologna, 2018.

[41] Kevin Purnell and Rolf Schwitter. Towards declarative smart contracts. In *The 4th Symposium on Distributed Ledger Technology*, pages 1–4. Griffith University, 05 2019.

[42] Jingwen Hu and Yong Zhong. A method of logic-based smart contracts for blockchain system. In *Proceedings of the International Conference on Data Processing and Applications*, ICDPA 2018, page 58–61, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364188. doi: 10.1145/3224207.3224218. URL https://doi.org/10.1145/3224207.3224218.

[43] Elisa Bertino, Barbara Catania, Vincenzo Gervasi, and Alessandra Raffaetà. Active-u-datalog: Integrating active rules in a logical update language. In Burkhard Freitag, Hendrik Decker, Michael Kifer, and Andrei Voronkov, editors, *Transactions and Change in Logic Databases*, pages 107–133, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-49449-2.

[44] Adriana Stancu and Mihaita Dragan. Logic-based smart contracts. In Álvaro Rocha, Hojjat Adeli, Luís Paulo Reis, Sandra Costanzo, Irena Orovic, and Fernando Moreira, editors, *Trends and Innovations in Information Systems and Technologies*, pages 387–394, Cham, 2020. Springer International Publishing. ISBN 978-3-030-45688-7.

[45] BigchainDB GmbH. Bigchaindb 2.0 the blockchain database. https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf, May 2018.

[46] leonardoalt. Logikon. https://github.com/logikon-lang/design, 2018. [Accessed 20th January 2022].

[47] Ethereum. Yul. https://docs.soliditylang.org/en/latest/yul.html, 2016. [Accessed 10th June 2022].

[48] Robert Kowalski and Fariba Sadri. Reactive computing as model generation. *New Generation Computing*, 33(1):33–67, Jan 2015. ISSN 1882-7055. doi: 10.1007/s00354-015-0103-z. URL https://doi.org/10.1007/s00354-015-0103-z.

[49] Imperial College London. Lps. http://lps.doc.ic.ac.uk, 2017. [Accessed 21st January 2022].

[50] Virgil Griffith and Vikram Verma. Designs for the l4 contract programming language based on deontic modal logic. https://archive.devcon.org/archive/watch/2/designs-for-the-l4-contract-programming-language-based-on-deontic-modal-logic, September 2016. [Accessed 20th June 2022].

[51] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory. The british nationality act as a logic program. *Commun. ACM*, 29(5):370–386, may 1986. ISSN 0001-0782. doi: 10.1145/5689.5920. URL https://doi.org/10.1145/5689.5920.

[52] Robert Kowalski and Akber Datoo. Logical english meets legal english for swaps and derivatives. *Artificial Intelligence and Law*, 30(2):163–197, Jun 2022. ISSN 1572-8382. doi: 10.1007/s10506-021-09295-3. URL https://doi.org/10.1007/s10506-021-09295-3.

[53] Florian Idelberger. Merging traditional contracts (or law) and (smart) e-contracts – a novel approach. In *The 1st Workshop on Models of Legal Reasoning*, 2020. URL https://lawgorithm.com.br/wp-content/uploads/2020/09/MLR2020-Florian-Idelberger.pdf.

[54] Keith L. Clark. *Negation as Failure*, pages 293–322. Springer US, Boston, MA, 1978. ISBN 978-1-4684-3384-5. doi: 10.1007/978-1-4684-3384-5_11. URL https://doi.org/10.1007/978-1-4684-3384-5_11.

[55] gRPC Authors. About grpc. https://grpc.io/about/, 2022. [Accessed 14th June 2022].

[56] Tendermint Inc. Iavl+ tree. https://github.com/cosmos/iavl, 2022. [Accessed 14th June 2022].

[57] Google Developers. Protocol buffers. https://developers.google.com/protocol-buffers, 2022. [Accessed 18th June 2022].

[58] Laura Foggan and Christine Elaine Cwiertny. Blockchain, smart contracts and parametric insurance: Made for each other. https://www.crowell.com/NewsEvents/Publications/Articles/Blockchain-Smart-Contracts-And-Parametric-Insurance-Made-For-Each-Other-1526190, 2018. [Accessed 8th June 2022].

[59] Rohini Sengupta and Carolyn Kousky. Parametric insurance for disasters. https://riskcenter.wharton.upenn.edu/issue-briefs/, 2020. [Accessed 8th June 2022].

[60] National Association of Insurance Commissioners. Parametric disaster insurance. https://content.naic.org/cipr-topics/parametric-disaster-insurance, 2022. [Accessed 8th June 2022].

[61] Japan Meteorological Agency. Tropical cyclone information : Scale and intensity of the tropical cyclone. https://www.data.jma.go.jp/multi/cyclone/cyclone_caplink.html?lang=en, 2015. [Accessed 8th June 2022].

[62] Tsung-Ting Kuo, Tyler Bath, Shuaicheng Ma, Nicholas Pattengale, Meng Yang, Yang Cao, Corey M. Hudson, Jihoon Kim, Kai Post, Li Xiong, and Lucila Ohno-Machado. Benchmarking blockchain-based gene-drug interaction data sharing methods: A case study from the idash 2019 secure genome analysis competition blockchain track. *International Journal of Medical Informatics*, 154:104559, 2021. ISSN 1386-5056. doi: https://doi.org/10.1016/j.ijmedinf.2021.104559. URL https://www.sciencedirect.com/science/article/pii/S1386505621001854.

[63] Gamze Gürsoy, Charlotte M. Brannon, and Mark Gerstein. Using ethereum blockchain to store and query pharmacogenomics data via smart contracts. *BMC Medical Genomics*, 13(1): 74, Jun 2020. ISSN 1755-8794. doi: 10.1186/s12920-020-00732-x. URL https://doi.org/10.1186/s12920-020-00732-x.

[64] Sai Batchu, Owen S. Henry, and Abraham A. Hakim. A novel decentralized model for storing and sharing neuroimaging data using ethereum blockchain and the interplanetary file system. *International Journal of Information Technology*, 13(6):2145–2151, Dec 2021. ISSN 2511-2112. doi: 10.1007/s41870-021-00746-3. URL https://doi.org/10.1007/s41870-021-00746-3.

[65] Sai Batchu, Karan Patel, Owen S Henry, Aleem Mohamed, Ank A Agarwal, Henna Hundal, Aditya Joshi, Sankeerth Thoota, and Urvish K Patel. Using ethereum smart contracts to store and share covid-19 patient data. *Cureus*, 14(1):e21378, January 2022. ISSN 2168-8184. doi: 10.7759/cureus.21378. URL https://europepmc.org/articles/PMC8853077.

[66] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph Liu, and Robin Doss. Security analysis methods on ethereum smart contract vulnerabilities: A survey, 2019. URL https://arxiv.org/abs/1908.08605.

[67] Phitchayaphong Tantikul and Sudsanguan Ngamsuriyaroj. Exploring vulnerabilities in solidity smart contract. In *ICISSP*, pages 317–324, 2020.

[68] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Comput. Surv.*, 53(3), jun 2020. ISSN 0360-0300. doi: 10.1145/3391195. URL https://doi.org/10.1145/3391195.

[69] Bitcoin Wiki. Secp256k1. https://en.bitcoin.it/wiki/Secp256k1, 2019. [Accessed 18th June 2022].

[70] Giovanni Ciatto, Roberta Calegari, S. Mariani, Enrico Denti, and Andrea Omicini. From the blockchain to logic programming and back: Research perspectives. In *WOA*, pages 69–74, 2018.

[71] Charlie Bowles. Smart contracts – legally enforceable? https://emlaw.co.uk/smart-contracts-legally-enforceable/, 2020. [Accessed 19th June 2022].

[72] Law Comission. Smart legal contracts: advice to government. https://www.lawcom.gov.uk/project/smart-contracts/, 2021. [Accessed 19th June 2022].

[73] Ethereum. Erc-20 token standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/, 2022. [Accessed 19th June 2022].

Appendix A

Full Code Listings

A.1 Crowdfunding

A.1.1 Chainlog

```
beneficiary('chainlog1n0tekhkd77ttvpac52jst8764l2qm952ghygvz').
deadline(1656633600).
goal(100000).

past_deadline :- time(Now), deadline(Deadline), Now > Deadline.
reached_goal :- total_raised(Total), goal(Goal), Total >= Goal.

state(fundraising)  :- not past_deadline.
state(unsuccessful) :- past_deadline, not reached_goal.
state(successful)   :- past_deadline, reached_goal, not claimed.
state(closed)       :- claimed.

total_raised(Total) :- sum_all(Contribution, funded(_, Contribution), Total).


on contribute:
  require state(fundraising);
  if      sender(Funder), value(Contribution);
  do      assert(funded(Funder, Contribution)).

on claimFunds:
  require state(successful);
  require sender(Sender), beneficiary(Sender);
  if      total_raised(Total);
  do      assert(claimed),
          transfer(Sender, Total).

on refund:
  require state(unsuccessful);
  if      sender(Sender),
          sum_all(Contribution, funded(Sender, Contribution), Total);
  do      retract(funded(Sender, _)),
          transfer(Sender, Total).
```

Listing A.1: Chainlog Crowdfunding Contract

A.1.2 Solidity – Simple

```
pragma solidity ^0.8;

contract CrowdFunding {

    address public constant BENEFICIARY = 0xAa596dFfeA2f94668A2E667Ced218442cA3F10DE;
    uint public constant DEADLINE = 1656633600;
    uint public constant GOAL = 100 ether;

```

```
9       enum State {
10          FUNDRAISING,
11          UNSUCCESSFUL,
12          SUCCESSFUL,
13          CLOSED
14      }
15
16      State public state = State.FUNDRAISING;
17      mapping(address => uint) public contributions;
18      uint public totalRaised;
19
20      function contribute() external payable {
21          updateState();
22          require(state == State.FUNDRAISING, "campaign not in fundraising state");
23          contributions[msg.sender] += msg.value;
24          totalRaised += msg.value;
25      }
26
27      function claimFunds() external {
28          updateState();
29          require(state == State.SUCCESSFUL, "campaign not in successful state");
30          require(msg.sender == BENEFICIARY, "caller is not the beneficiary");
31          state = State.CLOSED;
32          payable(msg.sender).transfer(totalRaised);
33      }
34
35      function refund() external {
36          updateState();
37          require(state == State.UNSUCCESSFUL, "campaign not in unsuccessful state");
38          uint contribution = contributions[msg.sender];
39          totalRaised -= contribution;
40          delete contributions[msg.sender];
41          payable(msg.sender).transfer(contribution);
42      }
43
44      function updateState() public {
45          if (state == State.CLOSED) {
46              return;
47          }
48          if (block.timestamp > DEADLINE) {
49              if (totalRaised < GOAL) {
50                  state = State.UNSUCCESSFUL;
51              } else {
52                  state = State.SUCCESSFUL;
53              }
54          }
55      }
56  }
```

Listing A.2: Solidity Crowdfunding Contract – Simple

A.1.3 Solidity – Extended

```
1   pragma solidity ^0.8;
2
3   contract CrowdFunding {
4
5       address public constant BENEFICIARY = 0xAa596dFfeA2f94668A2E667Ced218442cA3F10DE;
6       uint public constant DEADLINE = 1656633600;
7       uint public constant GOAL = 100 ether;
8
9       enum State {
10          FUNDRAISING,
11          UNSUCCESSFUL,
12          SUCCESSFUL,
13          CLOSED
14      }
15
16      State public state = State.FUNDRAISING;
17      mapping(address => uint) public contributions;
18      address[] public funders;
19      uint public totalRaised;
20
```

```
21      function contribute() external payable {
22          updateState();
23          require(state == State.FUNDRAISING, "campaign not in fundraising state");
24          contributions[msg.sender] += msg.value;
25          totalRaised += msg.value;
26          funders.push(msg.sender);
27      }
28  
29      function claimFunds() external {
30          updateState();
31          require(state == State.SUCCESSFUL, "campaign not in successful state");
32          require(msg.sender == BENEFICIARY, "caller is not the beneficiary");
33          state = State.CLOSED;
34          payable(msg.sender).transfer(totalRaised);
35      }
36  
37      function refund() external {
38          updateState();
39          require(state == State.UNSUCCESSFUL, "campaign not in unsuccessful state");
40          uint contribution = contributions[msg.sender];
41          totalRaised -= contribution;
42          delete contributions[msg.sender];
43  
44          // Remove sender from funders array
45          for (uint i = 0; i < funders.length; i++) {
46              if (funders[i] == msg.sender) {
47                  // Delete by replacing with last
48                  funders[i] = funders[funders.length - 1];
49                  funders.pop();
50              }
51          }
52  
53          payable(msg.sender).transfer(contribution);
54      }
55  
56      function updateState() public {
57          if (state == State.CLOSED) {
58              return;
59          }
60          if (block.timestamp > DEADLINE) {
61              if (totalRaised < GOAL) {
62                  state = State.UNSUCCESSFUL;
63              } else {
64                  state = State.SUCCESSFUL;
65              }
66          }
67      }
68  
69      function getFunders() external view returns (address[] memory) {
70          return funders;
71      }
72  
73      function getFundersAboveX(uint x) external view returns (address[] memory) {
74          address[] memory fundersAboveX = new address[](funders.length);
75          uint nextIndex = 0;
76          for (uint i = 0; i < funders.length; i++) {
77              address funder = funders[i];
78              if (contributions[funder] > x) {
79                  fundersAboveX[nextIndex++] = funder;
80              }
81          }
82          address[] memory result = new address[](nextIndex);
83          for (uint i = 0; i < nextIndex; i++) {
84              result[i] = fundersAboveX[i];
85          }
86          return result;
87      }
88  }
```

Listing A.3: Solidity Crowdfunding Contract – Extended

A.2 Licence

A.2.1 Chainlog

```
arbiter('chainlog1flkj3fyy8px5496te0gy82khw5ddv7u8y5tk0x').

action(A) :- member(A, [use, publish, comment, remove]).

forbidden(A) :- action(A), not permission(A).
permission(A) :- action(A), obligation(A).

% Article 1
permission(use) :- has_license.

% Article 2
permission(publish) :- has_license, has_approval.
obligation(remove) :- forbidden(publish), publish.

% Article 3
permission(comment) :- permission(publish).

% Article 4
obligation(publish) :- has_license, is_commissioned.

% Article 5
violation :- action(A), forbidden(A), A.
violation :- action(A), obligation(A), not A.
terminated :- violation.


% Message handlers
on grantLicense:   require arbiter; do assert(has_license).
on grantApproval:  require arbiter; do assert(has_approval).
on commission:     require arbiter; do assert(is_commissioned).

on declareUse:     require arbiter; do assert(use).
on declarePublish: require arbiter; do assert(publish).
on declareComment: require arbiter; do assert(comment).
on declareRemove:  require arbiter; do assert(remove).

arbiter :- sender(Sender), arbiter(Sender).
```

Listing A.4: Chainlog Licence Contract

A.2.2 Solidity

```
pragma solidity ^0.8;

contract License {
    address public constant ARBITER = 0x8018BB94ed02632ED1cc95d2aB10484edebBd5f0;

    bool public hasLicense;
    bool public hasApproval;
    bool public isCommissioned;

    bool public use;
    bool public usePermission;
    bool public useForbidden;

    bool public publish;
    bool public publishPermission;
    bool public publishForbidden;
    bool public publishObligation;

    bool public comment;
    bool public commentPermission;
    bool public commentForbidden;

    bool public remove;
    bool public removeObligation;

```

```
26     bool public violation;
27     bool public terminated;
28 
29     constructor() {
30         useForbidden = true;
31         publishForbidden = true;
32         commentForbidden = true;
33     }
34 
35     function evaluateLicenseContract() public {
36         // Article 1
37         if (hasLicense) {
38             useForbidden = false;
39             usePermission = true;
40         }
41         // Articles 2 and 4
42         if (hasLicense && (hasApproval || isCommissioned)) {
43             publishForbidden = false;
44             publishPermission = true;
45         }
46         // Article 2
47         if (hasLicense && !hasApproval && !isCommissioned && publish) {
48             removeObligation = true;
49         }
50         // Article 3
51         if (publishPermission) {
52             commentForbidden = false;
53             commentPermission = true;
54         }
55         // Article 4
56         if (hasLicense && isCommissioned) {
57             publishForbidden = false;
58             publishPermission = true;
59             publishObligation = true;
60         }
61         // Article 5
62         if (useForbidden && use
63             || publishForbidden && publish
64             || publishObligation && !publish
65             || commentForbidden && comment
66             || removeObligation && !remove) {
67             violation = true;
68             terminated = true;
69         }
70     }
71 
72     modifier onlyArbiter() {
73         require(msg.sender == ARBITER);
74         _;
75     }
76 
77     function grantLicense() external onlyArbiter {
78         hasLicense = true;
79         evaluateLicenseContract();
80     }
81 
82     function grantApproval() external onlyArbiter {
83         hasApproval = true;
84         evaluateLicenseContract();
85     }
86 
87     function commission() external onlyArbiter {
88         isCommissioned = true;
89         evaluateLicenseContract();
90     }
91 
92     function declareUse() external onlyArbiter {
93         use = true;
94         evaluateLicenseContract();
95     }
96 
97     function declarePublish() external onlyArbiter {
98         publish = true;
```

```
        evaluateLicenseContract();
    }

    function declareComment() external onlyArbiter {
        comment = true;
        evaluateLicenseContract();
    }

    function declareRemove() external onlyArbiter {
        remove = true;
        evaluateLicenseContract();
    }
}
```

Listing A.5: Solidity Licence Contract

A.3 Parametric Insurance

A.3.1 Chainlog

```
policyholder('chainlog103v2kw0xnhdfrnzukphheq9zlm67xg8ykjhfrc').
data_source('chainlog1rdhj3kcg9kr9y7pm489z579n9eqcsqyq0rjmwd').
insurer('chainlog12t7x9c08wyuurnw7ayhjvv7ycev29tstudcxrp').

policy_expiration(1672534800).

payout_limit(cyclone, 100).
payout_limit(earthquake, 100).
payout_limit(flood, 10).

% Typhoon policy
cyclone_category(violent_typhoon, T)  :- wind_speed(Speed, T), Speed >= 105.
cyclone_category(v_strong_typhoon, T) :- wind_speed(Speed, T), Speed >= 85, Speed < 105.
cyclone_category(typhoon, T)          :- wind_speed(Speed, T), Speed >= 64, Speed < 85.

payout_percentage(cyclone, 100, T) :- cyclone_category(violent_typhoon, T).
payout_percentage(cyclone, 80, T)  :- cyclone_category(v_strong_typhoon, T).
payout_percentage(cyclone, 40, T)  :- cyclone_category(typhoon, T).

% Earthquake policy
payout_percentage(earthquake, 100, T) :- seismic_intensity(Intensity, T),
                                         (Intensity = '6+' or
                                          Intensity = '7').
payout_percentage(earthquake, 80, T)  :- seismic_intensity(Intensity, T), Intensity = '6-'.
payout_percentage(earthquake, 40, T)  :- seismic_intensity(Intensity, T), Intensity = '5+'.

% Flood policy
ten_day_rainfall(TenDayMM, T) :-
  rainfall(_, T),
  add_time(T, days(10), T2),
  sum_all(OneDayMM, (rainfall(OneDayMM, Day), Day >= T, Day < T2), TenDayMM).

payout_percentage(flood, 100, T) :- ten_day_rainfall(TenDayMM, T), TenDayMM > 800.

% Eligibility to claim payout
eligible_for(Disaster, PercentagePayout) :-
  payout_percentage(Disaster, PercentagePayout, T),
  not (claimed(Disaster, ClaimT),
       add_time(ClaimT, weeks(4), T1),
       T < T1).

% Message handlers
on inputDisasterData(Data):
  require sender(Sender), data_source(Sender);
  do      assert(Data).

on claimPayout(Disaster):
  require sender(Sender), policyholder(Sender);
  require time(Now), policy_expiration(ExpT), Now < ExpT;
  require eligible_for(Disaster, PercentagePayout);

```

```
52  if      payout_limit(Disaster, Limit),
53          Payout is Limit * PercentagePayout / 100;
54
55  do      assert(claimed(Disaster, Now)),
56          transfer(Sender, Payout).
57
58 on terminate:
59  require time(T), policy_expiration(ExpirationDate), T > ExpirationDate;
60  if      insurer(Insurer), balance(Balance);
61  do      transfer(Insurer, Balance).
```

Listing A.6: Chainlog Parametric Insurance Contract

A.3.2 Solidity

```
1 pragma solidity ^0.8;
2
3 contract Insurance {
4
5     address public constant POLICYHOLDER = 0x4A02caF9c411332F5917af19289FDC74A6fB079c;
6     address public constant DATA_SOURCE = 0xF237A1826aA6Fc931B04bDeE1536224BF7E63C90;
7     address public constant INSURER = 0x0460c3fb9ce497308777020199Eb553F31E1bcE7;
8
9     uint public constant POLICY_EXPIRATION = 1672534800;
10
11     uint public constant CYCLONE_PAYOUT_LIMIT = 100 ether;
12     uint public constant EARTHQUAKE_PAYOUT_LIMIT = 100 ether;
13     uint public constant FLOOD_PAYOUT_LIMIT = 10 ether;
14
15     uint public cyclonePayoutPercentage = 0;
16     uint public earthquakePayoutPercentage = 0;
17     uint public floodPayoutPercentage = 0;
18
19     uint public nextCycloneEligibleTime;
20     uint public nextEarthquakeEligibleTime;
21     uint public nextFloodEligibleTime;
22
23     uint internal constant VIOLENT_TYPHOON_WIND_SPEED = 105;
24     uint internal constant V_STRONG_TYPHOON_WIND_SPEED = 85;
25     uint internal constant TYPHOON_WIND_SPEED = 64;
26
27     uint[10] internal rainfallRecordings;
28     uint internal nextRainfallIndex = 0;
29     uint internal rainfall10DaySum = 0;
30
31     constructor() payable {}
32
33     modifier onlyDataSource() {
34         require(msg.sender == DATA_SOURCE, "caller is not the data source");
35         _;
36     }
37
38     modifier onlyPolicyholder() {
39         require(msg.sender == POLICYHOLDER, "caller is not the policyholder");
40         _;
41     }
42
43     modifier beforeExpiration() {
44         require(block.timestamp < POLICY_EXPIRATION, "policy has expired");
45         _;
46     }
47
48     function inputWindSpeedData(uint speed, uint timestamp) external onlyDataSource {
49         // Ignore data if not eligible for cyclone payout yet.
50         if (timestamp < nextCycloneEligibleTime) {
51             return;
52         }
53
54         // Compare speed against policy parameters.
55         if (speed >= VIOLENT_TYPHOON_WIND_SPEED) {
56             cyclonePayoutPercentage = 100;
57         } else if (speed >= V_STRONG_TYPHOON_WIND_SPEED) {
58             if (cyclonePayoutPercentage < 80) {
```

```
59                cyclonePayoutPercentage = 80;
60            }
61        } else if (speed >= TYPHOON_WIND_SPEED) {
62            if (cyclonePayoutPercentage < 40) {
63                cyclonePayoutPercentage = 40;
64            }
65        }
66    }
67
68    function inputSeismicIntensityData(string calldata intensity, uint timestamp) external
       onlyDataSource {
69        // Ignore data if not eligible for earthquake payout yet.
70        if (timestamp < nextEarthquakeEligibleTime) {
71            return;
72        }
73
74        // Compare intensity against policy parameters.
75        if (stringsEqual(intensity, "6+") || stringsEqual(intensity, "7")) {
76            earthquakePayoutPercentage = 100;
77        } else if (stringsEqual(intensity, "6-")) {
78            if (earthquakePayoutPercentage < 80) {
79                earthquakePayoutPercentage = 80;
80            }
81        } else if (stringsEqual(intensity, "5+")) {
82            if (earthquakePayoutPercentage < 40) {
83                earthquakePayoutPercentage = 40;
84            }
85        }
86    }
87
88    function inputRainfallData(uint rainfall, uint timestamp) external onlyDataSource {
89        // Ignore data if not eligible for flood payout yet.
90        if (timestamp < nextFloodEligibleTime) {
91            return;
92        }
93
94        // Update the circular queue and running 10 day sum.
95        rainfall10DaySum = rainfall10DaySum + rainfall - rainfallRecordings[nextRainfallIndex];
96        rainfallRecordings[nextRainfallIndex] = rainfall;
97        nextRainfallIndex = (nextRainfallIndex + 1) % 10;
98
99        // Compare 10 day sum against policy parameter.
100       if (rainfall10DaySum > 800) {
101           floodPayoutPercentage = 100;
102       }
103   }
104
105   function claimCyclonePayout() external onlyPolicyholder beforeExpiration {
106       require(cyclonePayoutPercentage > 0, "not eligible for cyclone payout");
107       uint payout = CYCLONE_PAYOUT_LIMIT * cyclonePayoutPercentage / 100;
108
109       nextCycloneEligibleTime = block.timestamp + 4 weeks;
110       cyclonePayoutPercentage = 0;
111
112       payable(msg.sender).transfer(payout);
113   }
114
115   function claimEarthquakePayout() external onlyPolicyholder beforeExpiration {
116       require(earthquakePayoutPercentage > 0, "not eligible for earthquake payout");
117       uint payout = EARTHQUAKE_PAYOUT_LIMIT * earthquakePayoutPercentage / 100;
118
119       nextEarthquakeEligibleTime = block.timestamp + 4 weeks;
120       earthquakePayoutPercentage = 0;
121
122       payable(msg.sender).transfer(payout);
123   }
124
125   function claimFloodPayout() external onlyPolicyholder beforeExpiration {
126       require(floodPayoutPercentage > 0, "not eligible for flood payout");
127
128       nextFloodEligibleTime = block.timestamp + 4 weeks;
129       floodPayoutPercentage = 0;
130       delete rainfallRecordings;
```

```
131         rainfall10DaySum = 0;
132 
133         payable(msg.sender).transfer(FLOOD_PAYOUT_LIMIT);
134     }
135 
136     function terminate() external {
137         require(msg.sender == INSURER, "caller is not the insurer");
138         require(block.timestamp > POLICY_EXPIRATION, "policy has not expired yet");
139 
140         payable(msg.sender).transfer(address(this).balance);
141     }
142 
143     function stringsEqual(string calldata s1, string memory s2) internal pure returns (bool) {
144         return keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2));
145     }
146 }
```

Listing A.7: Solidity Parametric Insurance Contract

A.4 Pharmacogenomics Data

A.4.1 Chainlog

```
1 query(Gene, Variant, Drug, Observations) :-
2   find_all(observation(Gene, Variant, Drug, Outcome, Relation, SideEffect),
3            observation(Gene, Variant, Drug, Outcome, Relation, SideEffect),
4            Observations).
5 
6 on insert(Gene, Variant, Drug, Outcome, Relation, SideEffect):
7   do assert(observation(Gene, Variant, Drug, Outcome, Relation, SideEffect)).
```

Listing A.8: Chainlog Pharmacogenomics Data Contract

A.4.2 Solidity

```
1 pragma solidity ^0.8;
2 
3 contract PharmacogenomicsData {
4 
5     struct Observation {
6         string gene;
7         uint variant;
8         string drug;
9         string outcome;
10        bool relation;
11        bool sideEffect;
12    }
13 
14    mapping(uint => Observation) public database;
15    uint public counter;
16 
17    mapping(string => uint[]) public geneMapping;
18    mapping(uint => uint[]) public variantMapping;
19    mapping(string => uint[]) public drugMapping;
20 
21    function insert(
22        string calldata gene,
23        uint variant,
24        string calldata drug,
25        string calldata outcome,
26        bool relation,
27        bool sideEffect
28    ) external {
29        geneMapping[gene].push(counter);
30        variantMapping[variant].push(counter);
31        drugMapping[drug].push(counter);
32        database[counter] = Observation(gene, variant, drug, outcome, relation, sideEffect);
33        counter++;
34    }
35 
```

```
36      function query(
37          string calldata gene,
38          string calldata variant,
39          string calldata drug
40      ) external view returns (Observation[] memory) {
41          uint[] memory idList = new uint[](counter);
42          uint matchCount = 0;
43          uint[] memory genes;
44          uint[] memory variants;
45          uint[] memory drugs;
46
47          // If database empty, return empty
48          if (counter == 0) {
49              return new Observation[](0);
50          }
51
52          // Check number of fields being searched by
53          uint len = 0;
54          if (!stringsEqual(gene, "*")) {
55              len++;
56              genes = geneMapping[gene];
57              if (genes.length == 0) {
58                  return new Observation[](0);
59              }
60          }
61          if (!stringsEqual(variant, "*")) {
62              len++;
63              (bool success, uint variantUint) = stringToUint(variant);
64              require(success, "variant is not a valid uint");
65              variants = variantMapping[variantUint];
66              if (variants.length == 0) {
67                  return new Observation[](0);
68              }
69          }
70          if (!stringsEqual(drug, "*")) {
71              len++;
72              drugs = drugMapping[drug];
73              if (drugs.length == 0) {
74                  return new Observation[](0);
75              }
76          }
77
78          if (len == 0) {
79              // All fields are wildcards; push all IDs in database
80              matchCount = counter;
81              for (uint i = 0; i < counter; i++) {
82                  idList[i] = i;
83              }
84          } else {
85              // Compute intersection of genes, variants and drugs
86              matchCount = computeIntersection(genes, variants, drugs, len, matchCount, idList);
87          }
88
89          Observation[] memory results = new Observation[](matchCount);
90          for (uint i = 0; i < matchCount; i++) {
91              results[i] = database[idList[i]];
92          }
93          return results;
94      }
95
96      function stringsEqual(
97          string calldata s1,
98          string memory s2
99      ) internal pure returns (bool) {
100         return keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2));
101     }
102
103     function stringToUint(
104         string calldata s
105     ) internal pure returns (bool success, uint result) {
106         bytes memory b = bytes(s);
107         uint j = 1;
108         for (uint i = b.length - 1; i >= 0 && i < b.length;) {
```

```
109             if (!(uint8(b[i]) >= 48 && uint8(b[i]) <= 57)) {
110                 return (false, 0);
111             }
112             result += j * (uint8(b[i]) - 48);
113             j *= 10;
114             unchecked {
115                 i--;
116             }
117         }
118         success = true;
119     }
120 
121     function computeIntersection(
122         uint[] memory genes,
123         uint[] memory variants,
124         uint[] memory drugs,
125         uint len,
126         uint matchCount,
127         uint[] memory idList
128     ) internal view returns (uint) {
129         uint minLength = counter;
130         uint minField = 3;
131         if (genes.length > 0 && genes.length <= minLength) {
132             minLength = genes.length;
133             minField = 0;
134         }
135         if (variants.length > 0 && variants.length <= minLength) {
136             minLength = variants.length;
137             minField = 1;
138         }
139         if (drugs.length > 0 && drugs.length <= minLength) {
140             minLength = drugs.length;
141             minField = 2;
142         }
143 
144         for (uint i = 0; i < minLength; i++) {
145             uint numFieldsMatched = 1;
146             if (minField == 0) {
147                 // Shortest is genes
148                 if (variants.length > 0) {
149                     for (uint j = 0; j < variants.length; j++) {
150                         if (genes[i] == variants[j]) {
151                             numFieldsMatched++;
152                             break;
153                         }
154                     }
155                 }
156                 if (drugs.length > 0) {
157                     for (uint j = 0; j < drugs.length; j++) {
158                         if (genes[i] == drugs[j]) {
159                             numFieldsMatched++;
160                             break;
161                         }
162                     }
163                 }
164                 if (numFieldsMatched == len) {
165                     idList[matchCount++] = genes[i];
166                 }
167             } else if (minField == 1) {
168                 // Shortest is variants
169                 if (genes.length > 0) {
170                     for (uint j = 0; j < genes.length; j++) {
171                         if (variants[i] == genes[j]) {
172                             numFieldsMatched++;
173                             break;
174                         }
175                     }
176                 }
177                 if (drugs.length > 0) {
178                     for (uint j = 0; j < drugs.length; j++) {
179                         if (variants[i] == drugs[j]) {
180                             numFieldsMatched++;
181                             break;
```

```
182                     }
183                 }
184             }
185             if (numFieldsMatched == len) {
186                 idList[matchCount++] = variants[i];
187             }
188         } else if (minField == 2) {
189             // Shortest is drugs
190             if (genes.length > 0) {
191                 for (uint j = 0; j < genes.length; j++) {
192                     if (drugs[i] == genes[j]) {
193                         numFieldsMatched++;
194                         break;
195                     }
196                 }
197             }
198             if (variants.length > 0) {
199                 for (uint j = 0; j < variants.length; j++) {
200                     if (drugs[i] == variants[j]) {
201                         numFieldsMatched++;
202                         break;
203                     }
204                 }
205             }
206             if (numFieldsMatched == len) {
207                 idList[matchCount++] = drugs[i];
208             }
209         }
210     }
211 
212     return matchCount;
213 }
214 }
```

Listing A.9: Solidity Pharmacogenomics Data Contract