Building a Blockchain Simulation using the Idris Programming Language

Qiutai Pan
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN, USA
qiutai.pan@vanderbilt.edu

Xenofon Koutsoukos
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN, USA
xenofon.koutsoukos@vanderbilt.edu

ABSTRACT

The primary aim of this work is to create a program simulating a private distributed blockchain using the functional programming language Idris. This simulation is implemented such that a rock-paper-scissors game can be played between any two users of the blockchain via the use of smart contracts. Our motivation is to assert, using relevant features of Idris, that such an implementation possesses some of the accepted properties of blockchains. This paper first presents some differences between our implementation and most real-world blockchains. Next the Idris language and some of its features are discussed, focusing on how the language is used to implement the simulation. Finally, the advantages and disadvantages of utilizing Idris instead of an imperative programming language are examined.

CCS CONCEPTS

• **Software and its engineering** → **Functional languages**; • Computing Methodologies → Distributed Computing Methodologies

KEYWORDS

Blockchains, Cryptographic Hash Functions, Smart Contracts, Idris, Functional Languages, Dependent Types

ACM Reference Format:

Qiutai Pan and Xenofon Koutsoukos. 2019. Building a Blockchain Simulation using the Idris Programming Language. In *2019 ACM Southeast Conference (ACMSE 2019), April 18-20, 2019, Kennesaw, GA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299815.3314456

1 INTRODUCTION

Blockchains are data structures with many potential applications across various domains (i.e. storing records of transactions). However, there have not been many blockchain implementations developed using a functional language. Thus, our goal is to create a working implementation of a private distributed blockchain using Idris. Then, we investigate if Idris's features can be used to assert common blockchain properties. Finally, we identify advantages and disadvantages of using Idris instead of an imperative programming language.

This paper presents a working simulation of a blockchain in Idris, and we are able to assert a necessary property of blockchains using the language's appropriate features. However, it is also established that the relative advantages of using Idris for such an application did not outweigh the deficiencies, most notably in terms of efficiency. We begin this paper with a basic description of blockchains, describing the components of a constituent block, and explaining some associated concepts (such as smart contracts). Then, the desired specifications of the blockchain simulation that is to be implemented are explained, as well as how it will use smart contracts to allow any two "users" to play a game of rock-paper-scissors (RPS). Some potential advantages of playing a game via a blockchain are listed, as well as some desirable properties that help guarantee security. Afterwards, we introduce the Idris programming language. We examine the shared properties of its class of language (pure functional) and some of its distinctive features. Then, the specific details of how the blockchain simulation program was implemented using Idris, as well as how to execute the program, are discussed. Finally, several observed advantages and disadvantages of using Idris to implement such a blockchain simulation are discoursed.

2 BLOCKCHAIN CONCEPTS

On a basic level, a blockchain is simply a data structure consisting of individual elements called blocks where each block comprises a stored datum (e.g., a series of transactions) and several others fields storing metadata about the block itself [6]. These metadata fields include a cryptographic hash of the contents of the block, a copy of the hash of the preceding block in the blockchain, a block number representing the position of the block within the blockchain, and an integer nonce [6]. Thus, every block in a blockchain is cryptographically linked with its preceding block and it is very difficult for a malicious entity to modify it. This is because it is nearly impossible to alter the datum stored in a block without causing its hash field to change. It will be easy to detect that the hash field of the altered block no longer matches the previous hash field of the subsequent block

ACMSE 2019, April 18–20, 2019, Kennesaw, GA, USA

ACM ISBN 978-1-4503-6251-1/19/04...$15.00
https://doi.org/10.1145/3299815.3314456

unless the attacker is somehow able to simultaneously alter all the blocks from that point forward [6]. The process of adding a new block to a blockchain is known as mining, and involves the block's nonce field. For mining, an entity simply guesses nonces until one is found that satisfies a particular desired condition; usually that when the nonce is used as input into a cryptographic hash function alongside all other non-hash fields of the desired block, the result is an output hash that is below a certain desired level [1]. That hash and the successful nonce are then set as the respective fields of the block and the block is added to the blockchain.

Typically, the term "blockchain" is used to refer to distributed blockchains that are spread among users within a network, all of whom have their own copy [7]. There is no central server that stores a single official version of the blockchain (decentralized). Instead, all of the blockchains within the network are considered valid [7]. This further increases the security of the blockchain, as even if one user manages to successfully modify his or her version of the blockchain, all other users will then be able to see that the altered blockchain no longer matches their own and thereby reject that particular version [7]. A distributed blockchain can be considered public, where any entity with an internet connection can access it and initiate the addition of a new block. Alternatively, a distributed blockchain can be private, in which access to the blockchain is limited to a set number of users [7]. Real-world distributed blockchains often have an associated cryptocurrency such as Bitcoin, and users then compete with each other to mine new blocks for the blockchain. The entity that finds a satisfactory hash for a new block first is usually rewarded with an amount of the associated cryptocurrency [1].

Another concept that is associated with blockchains is that of smart contracts. Gatteschi et. al define them as small programs residing on a blockchain that execute certain operations when certain conditions are met [4]. However, the term "smart contract" is more commonly used to refer to blockchain programs that encode conditions for the execution of agreements or transactions between users and then execute them should their requirements be fulfilled. This is all performed without the need for intermediary entities [5].

3 SPECIFICATIONS

Our goal is to implement a simulation of a private distributed blockchain in which the number of users is fixed. There is no associated cryptocurrency and the data stored in the blocks are strings rather than transaction lists. Due to these assumptions, there is no economic incentive for the "users" within the network to compete amongst each other to mine new blocks [3]. Instead, the user that initiates the addition of a new node also mines that node, and then all other users verify the correctness of the node through a consensus protocol. Ideally, such a protocol would be non-blocking and resistant to users not responding to messages.

The simulation must support any two users within the network to play a game of rock-paper-scissors (RPS) via the use of smart contracts associated with the blockchain. The definition of a smart contract described earlier applies despite the lack of actual transactions. Among other benefits, executing a simple game on a distributed blockchain using smart contracts forgoes the need for an outside referee to determine the outcome of the game, and prevents the result of any game from being changed due to the immutability of the blockchain.

There are a few constraints that the implementation should uphold for security. Since in any game, a block containing the play of the first player is added to the blockchain before the second player makes their play, the first player's play must be obfuscated in such a way that the second player will not be able to determine the play of the first player [8]. Furthermore, users should be identified by the blockchain in such a way that a user cannot change his or her identity or attempt to pose as another. However, this property is be practically assured simply by assuming one initiates an instance of the distributed blockchain simulation correctly.

4 THE IDRIS PROGRAMMING LANGUAGE

Idris is an experimental functional programming language, still in active development, whose source code can be found in a public GitHub repository. As it is purely functional, the execution of Idris programs consists of evaluating side-effect free functions rather than statements as in imperative languages [2]. Due to not having side effects, Idris functions cannot alter a program's state in any way, and thereby will always produce the same output if given the same set of inputs (referential transparency). Also, replacing a functional call with its calculated output cannot affect a program's execution [2]. Furthermore, Idris functions are first-class constructs; all of the general operations of the language can be applied to them. Among other things, this means that they can be passed as input into or returned from other functions [2].

Likewise, types in Idris are also first-class constructs, and can be manipulated, used, passed as arguments to functions, and returned from functions just like any other value [2]. Idris also has dependent types, a concept rarely seen even in other purely functional languages. Dependent types are simply types that are calculated based on an outside value [2]. For example, the Idris type Vect is a dependent type that depends on a natural number and another type and Vect 4 String is a type that specifies a vector (list) of 4 Strings. In our case, dependent types can be used to specify that a blockchain has nondecreasing size. Furthermore, the Idris compiler contains a totality checker that is able to determine in almost all cases if a function is total or not. A total function is one that is guaranteed to return a value of its specified return type in a finite amount of time regardless of the values of its input arguments [2].

5 BLOCKCHAIN IMPLEMENTATION IN IDRIS

5.1 Overview of Implementation

Although the primary goal is to implement a simulation of a distributed blockchain (UDP sockets are used), it is also possible for the program to simulate a simple solitary blockchain. In both

cases, a blockchain is represented internally as a Vect of Blocks, where a Block is a record containing 5 fields, shown below in Figure 1. Throughout the rest of this paper, when a reference is made to adding a string to a blockchain, this actually means adding a block to the blockchain whose dataField field is that desired string.

```
record Block where
        constructor CreateBlock
        block : Nat
        nonce : Integer
        dataField : String
        prevHash : Bits 128
        hash : Bits 128
```

Figure 1: The Fields of a Block Record and Their Types

The implementation can be found on GitHub at https://github.com/sciadopitys/Idris-Blockchain. One specifies simple or distributed simulation when the program first begins running. To begin a simple blockchain simulation, one types ":exec runProc (procMain [])" in a single terminal running the program in the Idris runtime environment. To start a distributed blockchain simulation, one must type the same line but with a nonempty array argument, on as many open terminals as the desired number of network "users". At each terminal, the first element of the input array argument should be the desired port number of the current user's socket, while the remaining elements should be the port numbers of all other users. It is henceforth assumed that anyone running the simulation will provide correct input array arguments at each terminal such that all users in the simulation have different port numbers and have received the port numbers of each other user.

5.2 Simple Blockchain Simulation

The program repeatedly prompts the single user for a command until the user has decided to quit the program. Here, the only supported commands are "add", "display", and "quit. The "quit" command simply ends the program. The "display" command prints the entire contents of the current blockchain to the terminal. This is done by implementing the Show interface, which contains a show function for creating a String representation of a type, for the Block record type. Finally, the "add" command results in a new block being added to the current blockchain, and it is based on a call of the mining function findNonceAndHash.

The function findNonceAndHash takes as input Bits 128 representations of the desired block and dataField fields of the block to be added, its desired prevHash field (already a Bits 128), and an Integer that is the current nonce to be tried (in the simple simulation, the first nonce to be tried is always one). The function creates a Bits 128 representation of the nonce, and then obtains a cryptographic hash of that along with the input fields. If this obtained hash is determined to be below a certain level, then the function returns a pair of the successful nonce and the obtained hash. Otherwise, it recursively calls itself, with the nonce that was just tried incremented by one as the current nonce parameter. Since it cannot be guaranteed that a satisfactory nonce will ever be found, this recursive function may not terminate.

5.3 Distributed Blockchain Simulation

The interface to the distributed blockchain simulation is the same as that for the simple simulation. However, at each open terminal representing a network user, there are several additional supported commands besides "add", "display", and "quit". The latter two commands act exactly as in the simple simulation, but the blockchain is displayed only for the user who requested "display" and likewise the program ends only for the user who requested "quit". For the "add" command, not only will the new blockchain containing the new block has to be obtained as before, but also the user seeking to add the block must initiate a two-part consensus protocol. The user in question first sends a string containing all fields of the newly added block to all other users and waits for them to respond. If every other user responds with a message of "yes", then the first user sends another message to all other users confirming the addition to the blockchain. However, if any user had responded with any non-"yes" message, then the original user sends a message to all other users stating to revert to the earlier blockchain without the added block.

A "receive" command must be given to all other users in order for them to participate in the consensus protocol. Upon being given a "receive" command, a user blocks until it receives a message on its socket. Then, it splits the received string into the fields of the desired new block using '+' as the separator character (thus for the distributed simulation, strings containing the '+' cannot be added to the blockchain). The user then adds a block containing the desired new string to its own blockchain, and verifies that all fields of this block are identical to the corresponding fields in the received message. If so, then it sends a "yes" message back to the sender. Otherwise, or if the original message was not in the correct format, it sends "no" back. In any case, the user then blocks until it receives another message from the original sender, and finally implements or discards the addition to the blockchain based on that message.

5.4 Smart Contracts

The "rock", "paper", and "scissors" can be used to play a game of RPS between any 2 distinct users, where each user is identified by their port number. These commands are examples of smart contracts. Whenever a user provides one of these commands, the last block of the current blockchain is obtained, and that block's dataField field is checked to see if it is in the format of a RPS play or not. If so, then the current user would be the second player of a game, and the port number (a part of the dataField) of the user who made the first play is compared with that of the current user. If the numbers are the same, then the blockchain remains unchanged, as a user is not allowed to play both sides of a game. Otherwise, the move of the first player is obtained, a winner is determined based on the standard rules of RPS, and a

string proclaiming which player won (and their port number) is added to the blockchain via calling the "add" command.

If the dataField field of the last block had not been in the format of a RPS play, then the current user must be the first player of a prospective game. In this case, an Integer value (the commit value) must have been provided along with the command to obfuscate the play of the first player and thereby prevent any second player from gaining an unfair advantage. A cryptographic hash of the Bits 128 representations of the play and the commit value is obtained and then converted back into a String. Finally, a String containing the port number of the current user, the commit value, and the String representation of the obtained cryptographic hash is added to the blockchain via calling the "add" command. The various sections of this String are separated by the '*' char. Thus for the distributed simulation, strings containing '*' cannot be added either.

6 ADVANTAGES AND DISADVANTAGES OF IDRIS

There are several reasons why implementing a blockchain simulation in Idris is beneficial. The most pressing advantage is due to the nature of simple blockchains; they are essentially linear data structures. Representing a blockchain in Idris using the dependent Vect data type not only captures this inherent structure, but also allows a user to easily establish constraints on the size of a blockchain. For example, using the Vect data type and a user-defined dependent type based on Vect, one can ensure a basic property about simple blockchains – nondecreasing size – just by setting the return types of relevant functions to the dependent type instead of Vect. This dependent type is shown below in Figure 2.

```
data VectSameOrInc : Type -> Type where
   Same : (len : Nat) -> Vect len a -> VectSameOrInc a
   Inc : (len : Nat) -> Vect (S len) a -> VectSameOrInc a
```

Figure 2: User-defined Data Type VectSameOrInc

Furthermore, the functional paradigm of pattern matching on a structure with zero, one, or more elements is more intuitive and elegant than performing traversals through a structure as would likely be done when using an imperative language. Additionally, both the simple and distributed blockchain simulations require I/O operations such as obtaining and processing user input. It is typically easy to make errors when coding I/O, but Idris mitigates this by enforcing separation of I/O and pure operations providing simple sequencing of I/O operations via the use of the monadic >>= operator or do blocks. It is also easy to call pure functions within sequences of I/O operations by making use of Idris's "pure" function. Also, the type checker included with the Idris compiler is can establish that all of the helper (non-mining) functions in the program are indeed total.

Unfortunately, there are also several substantial disadvantages. The most concerning disadvantage is the greatly decreased efficiency and increased running time of user commands. This is due to the nature of functional languages – since there are no variables, the blockchain often needs to be copied and passed to functions. This is inefficient, especially when the blockchain has grown in size. Furthermore, we are currently making use of the idris-crypto package found on GitHub, but unfortunately both the MD5 and SHA cryptographic hash function implementations in the package have unresolved issues. Therefore, the desired level in the mining function is currently set to be so high as to accept the very first nonce tried each time. Also, there is no timeout mechanism for sockets in Idris, so it is impossible to implement a non-blocking consensus protocol. Thus the simulation is indeed vulnerable to users simply not responding (it will block indefinitely). Additionally, due to the strict separation of I/O and pure code, one cannot easily print statements to the console for debugging purposes. Another potential issue is that Idris has meaningful whitespace. This is not present in most imperative languages and may lead to unusually wide lines of code. Finally, although defining functions using pattern matching is usually beneficial, in certain situations this may lead to one being unable to define a helper function to reduce redundancy.

7 CONCLUSION

Overall, attempting to implement a blockchain simulation in Idris will almost certainly be a beneficial experience for a programmer seeking to gain experience with functional programming languages and learn how to use dependent types. However, even though Idris does provide several advantages for those seeking to create a blockchain simulation, chief among them being the ability to guarantee desired properties as part of the program itself rather than via assertions or tests, these are likely not sufficient to ignore the poor efficiency and other drawbacks of the language for this purpose. Therefore, unless the disadvantages presented earlier are resolved with future updates to the language, implementing a serious blockchain simulation in Idris is not recommended.

REFERENCES

[1] N. Acheson. 2018. How Bitcoin Mining Works. (January 2018). https://www.coindesk.com/information/how-bitcoin-mining-works.
[2] E. Brady. 2017. *Type-driven Development with Idris*. Manning Publications, Shelter Island, NY.
[3] K. Christidis and M. Devetsikiotis. 2016. Blockchains and Smart Contracts for the Internet of Things. *IEEE Access* 4 (May 2016), 2292-2303. DOI: https://doi.org/10.1109/ACCESS.2016.2566339.
[4] V. Gatteschi, F. Lamberti, C. Demartini, C. Pranteda, and V. Santamaría. 2018. Blockchain and Smart Contracts for Insurance: Is the Technology Mature Enough? *Future Internet* 10, 2, Article 20 (February 2018), pp. 1-16. DOI: https://doi.org/10.3390/fi10020020
[5] T. Horda. 2018. A Guide to Smart Contracts and Their Implementation. (January 2018). https://rubygarage.org/blog/guide-to-smart-contracts.
[6] A. Lewis. 2015. A Gentle Introduction to Blockchain Technology – Bits on Blocks. https://bitsonblocks.net/2015/09/09/gentle-introduction-blockchain-technology/#more-72.
[7] M. H. Miraz and M. Ali. 2018. Applications of Blockchain Technology beyond Cryptocurrency. arXiv:1801.03528. https://arxiv.org/abs/1801.03528.
[8] J. Pettersson and R. Edström. 2016. *Safer smart Contracts through Type-driven Development*. Master's thesis. Chalmers University of Technology & University of Gothenburg, Gothenburg, Sweden.