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 purely functional programming language Idris. This simulation is to be implemented such that a simple 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 the unique features of the Idris language, that such an implementation possesses some of the accepted properties of blockchains. This paper begins by introducing concepts associated with blockchains and their definitions, along with some differences between this implementation and most real-world blockchains. Then, the Idris programming language and some of its relevant features are discussed, as well as how the language was used to implement the simulation. Finally, the advantages (i.e. dependent types) and disadvantages (i.e. efficiency) of utilizing such a language as opposed to an imperative programming language like C are discussed.

KEYWORDS

Blockchains, Mining, Cryptographic Hash Functions, Smart Contracts, Rock-Paper-Scissors, Idris, Functional Languages, Dependent Types, Totality

ACM Reference format:

Qiutai Pan and Xenofon Koutsoukos. 2018. Building a Blockchain Simulation using the Idris Programming Language. In *ACM SE ’19: ACM SE ’19: Southeast Conference, April 18-20, 2019, Kennesaw, GA, USA.* ACM, New York, NY, USA, 6 pages.

1 Introduction

Blockchains are recently invented data structures with many potential applications across a wide variety of spheres, but are most often used to store records of transactions. However, there have not been many prominent examples of an implementation of a blockchain developed in a functional language. Thus, we set out to create such an implementation using the language Idris and see if implementing blockchains in functional languages is a worthwhile endeavor. The goals of this work are threefold. First, to create a reasonably working implementation of a simulation of a distributed blockchain using the Idris programming language, according to the specifications described later in this section (it has some key differences from real-world distributed blockchains). Then, to discover if Idris language features can be used to assert common properties of blockchains. Finally, to identify the relative advantages and disadvantages of using Idris to implement a blockchain simulation instead of a more widely used imperative programming language. We are able to achieve all three of our stated goals - We developed a reasonably working simulation of a blockchain in Idris, and we were able to assert a necessary property of blockchains using the language’s unique features. However, it was also established that the relative advantages of using a functional language for such an application did not outweigh its deficiencies, most notably in terms of efficiency.

We begin this paper with giving the most basic definition of a blockchain, describing the components of a constituent block, and explaining concepts associated with these. Next, we define distributed blockchains and smart contracts, and provide real-world examples of each. Then, the desired specifications of the blockchain simulation that is to be implemented are explained, as well as how the simulation will use smart contracts to allow two users to play a game of rock-paper-scissors between them. Some potential advantages of playing a rock-paper-scissors game using a blockchain are listed, as well as a few desirable properties that help guarantee security.

Afterwards, we introduce the Idris programming language. The shared properties of its class of language – purely functional – are introduced and defined, as well as some of its distinctive features. These include dependent types and a totality checker. Then, the specific details of how the blockchain simulation program was implemented using Idris, as well as how to actually execute the program, are discussed. This is divided into 3 sections – how the program simulates a simple, single blockchain, how the program simulates a distributed blockchain, and how smart contracts are used to play rock-paper-scissors games. Finally, several determined advantages and disadvantages for using the Idris programming language to implement such a distributed blockchain simulation are discoursed, with examples provided for some of these points. We end with a brief discussion of another project that seeks to implement a blockchain simulation in the Idris programming language.

2 Blockchain Concepts

At its most basic definition, a blockchain is simply a data structure consisting of individual elements called blocks where each block comprises a stored datum (in most real-world blockchains, this is a series of transactions) and several others fields storing metadata about the block itself [10]. These metadata fields, shown below in figure 1, include a cryptographic hash of the contents of the current 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, an integer nonce, and often a timestamp[10].

Singh and Singh clarify that the very first block within a blockchain is usually referred to as the genesis block and generally has block number and pervious hash fields of zero [14]. Since every block in a blockchain is cryptographically linked with its preceding block, it is very difficult for a malicious entity to modify an existing block of a blockchain. This is because it is nearly impossible to alter the datum stored in a block without causing its hash field to change. Then, others will easily be able to see that the hash field of the altered block no longer matches the previous hash field of the next block unless the attacker is somehow able to simultaneously alter all the blocks from that point forward [10].

The process of adding a new block to a blockchain is usually referred to as mining, and involves the block’s nonce field. The idea of mining is quite simple – an entity simply guesses (usually randomly) nonces until one is found that satisfies a particular desired condition. This condition usually is 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 (this is generally determined by its number of leading zeros)[2]. That hash and the successful nonce are then set as the respective fields of the block and the block is added to the blockchain.

Miraz and Ali state that in most circumstances, the term “blockchain” is used to refer to distributed blockchains in which a blockchain is spread among all users within a network, all of whom have their own copy [11]. There is no central server that stores a single official version of the blockchain (the blockchain is decentralized). Instead, all of the blockchains within the network are considered valid [11]. This further increases the security of the blockchain, as even if one user manages to 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 [11].

Miraz and Ali also state that a distributed blockchain can be considered public, where any entity with an internet connection can access the blockchain and/or 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 [11]. In the real world, distributed blockchains often have an associated cryptocurrency such as Bitcoin or Ether, and users (“miners”) then compete with each other to mine new blocks for the blockchain. The entity that finds the appropriately leveled hash for a new block first is usually rewarded with an amount of the associated cryptocurrency [2].

Another concept that is intimately associated with blockchains is that of the smart contract. The most rudimentary definition of a smart contract, given by Gatteschi et al., is a small program residing on a blockchain that executes certain operations when certain conditions are met [7]. 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 those transactions should their requirements be fulfilled. This is all performed without the need for intermediary entities [8]. An example of this would be escrow, which runs on the Ethereum blockchain [6].

Figure 1 - A typical block without timestamp [4]

3 Specifications

Our goal is to implement a simulation of a private distributed blockchain in which the number of users is completely fixed (no users can join or leave the “network”). There is no associated cryptocurrency, and the data stored in the blocks are not lists of transactions, but instead just strings. Due to this, along with the fact that the simulated network is private rather than public, Christidis and Devetsikiotis state that there is no “economic incentive” for the “users” within the network to compete amongst each other to mine new blocks [5]. 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 newly added node through a consensus protocol. Ideally, such a consensus protocol would be non-blocking and resistant to malicious users deliberately not responding to messages from others.

The simulation is also implemented such that any two users within the “network” can play a simple game of rock-paper-scissors via the use of smart contracts associated with the blockchain. These games follow the officially defined rules of rock-paper-scissors without additional plays beyond the original three, where rock beats scissors, paper beats rock, scissors beats paper, and any play “stalemates against itself” [15]. The basic definition of a smart contract as established earlier applies despite the lack of actual transactions. Among other benefits, executing a rock-paper-scissors game on a distributed blockchain using smart contracts forgoes the need for an outside referee to determine the outcome of a game between two users, and prevents the result of any game from being changed due to the immutability of the blockchain(s).

There are a few constraints that the implementation should uphold in order for it to be considered secure. 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, Pettersson and Edström warn that the first player’s play must be cryptographically hashed in such a way that the second player will not be able to determine what the play of the first player was [12]. Furthermore, users should be identified by the blockchain in such a way that a user cannot somehow change his or her identity or attempt to masquerade as another. However, this property can be practically assured simply by carefully restricting how an instance of a distributed blockchain simulation is started.

4 The Idris Programing Language

Idris is an experimental functional programming language whose source code can be found in a public GitHub repository. It was created by Edwin Brady, and is still in active development. As it is a purely functional language, the execution of Idris programs consists of evaluating side-effect free functions rather than statements as in the more familiar imperative languages like C or Python [3]. 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 repeatedly given the same set of inputs (referential transparency). This means that if any function call is replaced with its calculated output, a program will execute exactly the same way [3]. Although pure functions cannot explicitly allow user interaction via console input and output [3], Idris allows functions to be written that describe a sequence of potentially impure IO operations. These functions must return a value of type IO *x*, where *x* is some type, and are considered by the compiler to still be pure [3].

Figure 2 - The fields of a Block record and their types

record Block where

constructor CreateBlock

block : Nat

nonce : Integer

dataField : String

prevHash : Bits 128

hash : Bits 128

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 [3]. Idris is similar in many ways, including basic syntax, to the more well-known pure functional language Haskell. However, the two languages do have some key differences. For example, Idris defaults to eager evaluation of function arguments, in which all of a function’s arguments are always evaluated before the function body. On the other hand, Haskell has lazy evaluation by default – function arguments are not evaluated until their values are explicitly needed [9].

An important feature of Idris not present in Haskell at this time is the use of types as a first-class construct [3]. Because of this, according to Brady (2017) Idris types “can be manipulated, used, passed as arguments to functions, and returned from functions just like any other value, such as numbers, string, or lists.” Idris also allows the use of 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 [3]. For example, the Idris type Vect is a dependent type that depends on a natural number and another type – Vect 4 String is a type that specifies a vector (list) of 4 Strings. In this simulation, dependent types can be used to specify that a blockchain is always growing or remaining the same size, and never decreasing in size.

Furthermore, the Idris compiler contains a totality checker that is able to determine in almost all cases if a particular function is total or not. A total function is a function 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 [3]. Unfortunately, this feature is not especially useful for our purposes as any functions involved in the process of mining a new block for a blockchain cannot be determined to be total – it simply cannot be guaranteed that a suitable nonce will ever be found.

5 Blockchain Implementation in Idris

The developed code is publicly available on GitHub at https://github.com/sciadopitys/Idris-Blockchain. It is possible for anyone to download and modify the base code, although there are currently a few limitations; we discuss these in a subsequent section. Although the primary goal is to implement a simulation of a distributed blockchain, 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 2.

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.

To begin an execution of an instance of the program at an open terminal, a user must type “idris –p contrib BlockchainMain.idr”. A user specifies what type of simulation he or she wants when the program first begins running. To begin a simple blockchain simulation, one simply types “:exec runProc (procMain [])” in a single terminal running the program in the Idris runtime environment. To start a distributed blockchain simulation (which uses UDP sockets to simulate communication between different “users”), one must type the same line but with a nonempty array argument, on as many open terminals running the program in the Idris runtime environment 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 indeed provide correct input array arguments at each terminal such that all users in the simulation have different port numbers and have received the correct port numbers of each other user. Furthermore, the program does not allow a user to provide a single-element input array; if this occurs, the program simply exits.

5.1 Simple Blockchain Simulation

For the simple blockchain simulation, a process is spawned that simulates the Idris replWith loop. This means that the program repeatedly prompts the single user for a command until the user has decided to quit the program. For the simple blockchain, the only supported commands are “add”, “display”, and “quit”; any other user input is not recognized (the user is then prompted again). The “quit” command simply causes the looping process to pass a message to the main process of the program stating that it has finished, and the entire program then immediately ends. The “display” command is also simple – it just prints the entire contents of the current blockchain (i.e. the fields of every block in the 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, which eventually results in a new block being added to the current blockchain, is somewhat more involved. A few function calls are performed, eventually leading to the initial call of the mining function, named findNonceAndHash.

The findNonceAndHash function takes as input Bits 128 representations of the desired block and dataField fields of the block to be added, its desired prevHash field (which is already a Bits 128), and an Integer that is the current nonce to be tried (for the simple simulation, the first nonce to be tried is always one). The function itself 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 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, passing the nonce that was just tried incremented by one for the current nonce parameter (and the same Bits 128’s for all other parameters).

Since it cannot be guaranteed that a satisfactory nonce will ever be found, this recursive function cannot be guaranteed to terminate and thereby cannot be total. Furthermore, we are currently making use of the idris-crypto package found on GitHub at https://github.com/idris-hackers/idris-crypto, 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. However, once the implementations in the package are fixed or some other solution is found, it would be straightforward to lower the desired level in the mining function.

5.2 Distributed Blockchain Simulation

The interface to the distributed blockchain simulation is essentially 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 “quit” and “display” commands behave exactly the same way as for the simple simulation. The blockchain is displayed only for the user who requested “display”, and likewise the program ends only for the user who requested “quit” (although then the consensus protocol unfortunately no longer works correctly for the remaining users; we explain this later). However, for the “add” command, not only will the new blockchain containing the new block have to be obtained as before, but also the user seeking to add the block must initiate the consensus protocol. This simulation uses the two-part consensus protocol. The user in question first sends a string containing all fields of the newly added block to all other users. Then, it waits for those other users to respond. If every other user responds with a message of “yes”, then the first user sends another message to all other users, this time confirming the addition to the blockchain. However, if any user had responded with a message of “no” (or something other than “yes”), then the original user sends a message to all other users stating to revert to the earlier blockchain without the added block.

The “receive” command must be given to all other users in order for them to participate in the consensus protocol. Upon acknowledging 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 (this means that for the distributed simulation, strings containing the plus character 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” confirmational message back to the sender. Otherwise, it sends “no” back. If the original message was not in the correct format, then the user will also respond back with a “no” denial. 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 what that message says. Unfortunately, since there is no timeout mechanism for sockets (UDP or TCP) in Idris, it is impossible to implement a non-blocking consensus protocol, and the simulation is indeed vulnerable to users simply not responding (i.e. not typing the “receive” command) and thereby causing the consensus protocol to block indefinitely.

5.3 Smart Contracts

There are also 3 commands – “rock”, “paper”, and “scissors” – that can be used to play a game of rock paper scissors between any 2 distinct users, where each user is identified by their port number. It is assumed that the person running the simulation has given each “user” a unique port number and that no user can spontaneously change his or her port number to be the same as that of some other user. Therefore, it is assumed that attacks in which a user impersonates someone else cannot occur. These commands are examples of the basic definition of smart contracts, described previously. 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 rock paper scissors play or not. If so, then the current user would be the second player of a rock paper scissors game, and the port number (which is part of the dataField) of the user who made the first play is compared with that of the current user. If the port numbers are the same, then the blockchain remains unchanged, as a user is not allowed to play both sides of a rock paper scissors game. Otherwise, the actual move of the first player is obtained, a winner is determined based on the standard rules of rock paper scissors, 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 rock paper scissors play, then the current user must be the first player of a prospective RPS game. In this case, an Integer value (to be henceforth called the commit value) must have been provided along with the command. This commit value is used to obfuscate the play of the first player and thereby prevent any potential second player from gaining an advantage due to being able to see the play of the first player. 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 (again via calling the “add” command). The various sections of this String are separated by the ‘\*’ char; this means that for the distributed simulation, strings containing the asterisk character cannot be added to the blockchain either. The code for this portion of the implementation of the “rock” command, with leading whitespace removed, is shown below in figure 3.

Figure 4 - User-defined Data Type VectSameOrInc

data VectSameOrInc : Type -> Type where

Same : (len : Nat) -> Vect len a -> VectSameOrInc a

Inc : (len : Nat) -> Vect (S len) a -> VectSameOrInc a

Nothing => pure (Just ("Commit value not an integer\n", (Same m chain)))

Just commitInt => let playToInt = getIntRep (unpack "rock") 0

playBits = the (Bits 128) (intToBits playToInt)

commitBits = the (Bits 128) (intToBits commitInt)

playHash = hashMessage dummyMD5 [playBits, commitBits]

playHashStr = bitsToStr playHash

in executeCommandAlt chain "add" (" " ++ (show port) ++ "\*" ++ commit ++ "\*" ++ playHashStr) sock addrs port

Figure 3 - Portion of "rock" command

6 Advantages and Disadvantages of Idris

There are several arguments for why implementing a blockchain simulation in a purely functional language with dependent types such as Idris would be beneficial. The most pressing advantage is due to the nature of simple blockchains, as they are essentially linear data structures. Representing a blockchain in Idris using the dependent Vect data type (as described earlier) not only captures this inherent structure of a simple blockchain, but also allows a user to establish more constraints on the size of a blockchain than would be possible in any language without dependent types. 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 – that their size is always nondecreasing – just by setting the return types of relevant functions to include the user-defined dependent type instead of simply Vect. This dependent type is shown below in figure 4.

Furthermore, since a simple blockchain is usually represented as a linear data structure, when working with a blockchain the functional paradigm of pattern matching on a blockchain with zero, one, or more elements can be considered more intuitive and elegant than performing a traversal through a blockchain representation 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. Therefore, the forced separation of I/O and pure operations in Idris (which is generally not enforced in imperative languages) likely aids the user with avoiding potential errors with I/O, such as those operations potentially modifying program state (in this case, this would just be the blockchain itself). Furthermore, Idris allows for easy sequencing of I/O operations via the use of the monadic >>= operator and/or do blocks (which is actually just syntactic sugar for repeated uses of >>=). In Idris, it is also quite simple to call pure functions within sequences of I/O operations by making use of the “pure” function, which simply returns a value of type IO x given an input of type x. Also, the type checker included with the Idris compiler, while unable to prove that the mining functions in the program (as well as any functions that would call them, and so forth) are total, is still useful to establish that all of the helper functions in the program are indeed total. Finally, if one uses the Atom text editor to develop in Idris, the language-idris package allows users to perform several interactive editing commands. For example, Ctrl-Alt-R immediately performs a type check of the current program.

For all of the strengths of Idris, there are also several substantial disadvantages to implementing a blockchain simulation in such a language. The most concerning disadvantage would be the greatly decreased efficiency and increased running time of user commands. Unfortunately, this is due to the nature of functional languages – since there are no variables as in imperative languages, the blockchain often needs to be copied and passed to functions. This effect is especially noticeable when the blockchain has grown in size. Furthermore, at the current point there is only one package in Idris with implementations of cryptographic hash functions, and neither the SHA nor MD5 implementations in the package work correctly. There is also no timeout mechanism for IPC with sockets, and so it is currently impossible to implement a non-blocking consensus protocol.

A perhaps less significant disadvantage is that due to the strict separation of I/O and pure code, one cannot easily print statements to the console for debugging purposes (as one often does in imperative languages) in functions. Furthermore, there is no actual debugger for the language, although there are some functions that can be used to debug a program. Another potential issue is that whitespace is meaningful in Idris. This feature is not present in most imperative languages – Python being a notable exception [1] – and may lead to unusually wide lines of code if one is adamant about using tabs instead of spaces. 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. This is the case for the “rock”, “paper”, and “scissors” smart contracts, which are all part of the same function definition and call that function definition for another command (the “add command). In any case, one must remember that Idris is an experimental language still in active development, and so it is certainly possible that some of the issues encountered may be resolved in the future.

7 Related Work

Currently, there appears to be only one other publically available Idris blockchain project on GitHub that contains actual code, found at https://github.com/rpeszek/idris-chain [13]. There are several key differences between that project and this one. The primary difference is in the choice of representation for simple blockchains – rpeszek’s project represents them as cryptographically linked lists rather than as Vects. It was not attempted to accurately determine the benefits and drawbacks of these differing implementations, or to decide which one is superior overall. Furthermore, the blockchains in rpeszek’s project are parameterized by type, and therefore blockchains containing data of different types can be constructed, whereas in this project all blockchain can only store Strings. In fact, rpeszek’s project even supports heterogenous blockchains whose constituent blocks themselves can store data of varying types. It also includes several tests for the rest of the project, a feature that was deemed unnecessary for this project. However, rpeszek’s project does not yet allow for the simulation of distributed blockchains (only simple blockchains have been implemented), and is currently using non-cryptographic hash functions rather than cryptographic, likely due to the aforementioned issues with the idris-crypto package.

8 Conclusion

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

REFERENCES

[1] Python Software Foundation. 2.1.7 Indentation. Retrieved from https://docs.python.org/2.0/ref/indentation.html.

[2] Noelle Acheson. 2018. How Bitcoin Mining Works. (January 2018). Retrieved from https://www.coindesk.com/information/how-bitcoin-mining-works/.

[3] Edwin Brady. 2017. *Type-driven Development with Idris*. Manning Publications, Shelter Island, NY.

[4] Anders Brownworth. Block. Retrieved from https://anders.com/blockchain/block.html.

[5] Konstantinos Christidis and Michael 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

[6] David Floyd. 2018. The Top 5 Ethereum Dapps By Daily Active Users. (June 2018). Retrieved from https://www.coindesk.com/top-5-ethereum-dapps-daily-active-users/.

[7] Valentina Gatteschi, Fabrizio Lamberti, Claudio Demartini, Chiara Pranteda, and Víctor Santamaría. 2018. Blockchain and Smart Contracts for Insurance: Is the Technology Mature Enough? *Future Internet* 10, 2, Article 20 (February 2018), 1-16. DOI: https://doi.org/10.3390/fi10020020

[8] Tania H. A Guide to Smart Contracts and Their Implementation. Retrieved from https://rubygarage.org/blog/guide-to-smart-contracts.

[9] Cornell University. 2004. Lecture 6: Eager vs. Lazy Evaluation. Higher-Order Functions. Retrieved from http://www.cs.cornell.edu/courses/cs312/2004fa/lectures/lecture6.html.

[10] Antony Lewis. 2015. A Gentle Introduction to Blockchain Technology – Bits on Blocks. (September 2015). Retrieved from https://bitsonblocks.net/2015/09/09/gentle-introduction-blockchain-technology/#more-72.

[11] Mahdi H. Miraz and Maaruf Ali. 2018. Applications of Blockchain Technology beyond Cryptocurrency. arXiv:1801.03528. Retrieved from https://arxiv.org/abs/1801.03528.

[12] Jack Pettersson and Robert Edström. 2016. *Safer smart contracts through type-driven development*. Master’s thesis. Chalmers University of Technology & University of Gothenburg, Gothenburg, Sweden.

[13] Rpeszek. 2018. Rpeszek/idris-chain. (June 2018). Retrieved from https://github.com/rpeszek/idris-chain.

[14] Sachchidanand Singh and Nirmala Singh. 2016. Blockchain: Future of financial and cyber security. *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)* (December 2016), 463-467.

[15] World Rock Paper Scissors Society. 2015. The Official Rules of Rock Paper Scissors. (May 2015). Retrieved from http://worldrps.com/game-basics/the-official-rules-of-rock-paper-scissors/.