Specification of Bitcoins and other Cryptocurrencies

Kiruthesh Pugalenthy
1913621

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Bachelor of Science

Bachelor of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

May 03, 2022

Declaration

This work has not been previously accepted in substance for any degree and is not being con- currently submitted in candidature for any degree.

Signed Kiruthesh Pugalenthy

Date 03/05/2022

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed Kiruthesh Pugalenthy

Date 03/05/2022

Statement 2

I hereby give my consent for my thesis, if accepted, to be made available for photocopying and inter-library loan, and for the title and summary to be made available to outside organisations.

Signed Kiruthesh Pugalenthy

Date 03/05/2022

I would like to dedicate this work to Santoshi Nakamoto for revolutionising cryptocurrencies

Abstract

We discuss fundamental cryptocurrency concepts that are crucial in order to understand cryptocurrencies. In particular we look at consensus mechanisms, the background of cryptocurrencies, blockchain technology and drawback of cryptocurrencies.

We look at majority of exploits and attacks and how such attacks are avoided. I also touch on cryptography in cryptocurrencies and its significance.

We also look at different languages used for modelling and compare it the Agda. Looking at the advantages and disadvantages of Agda. Other more well-known languages exist for proofs and modelling such as Coq, however Coq is more proof focused.

This study is carried out to identify the need for greater verification techniques and proofs, to validate the concept of cryptocurrencies and Bitcoin.

The practical element of this study is modelling the bitcoin ledger in functional programming language Agda. We focus on modelling transactions for each block and demonstrate the interaction between addresses while having several inputs and outputs. We also prove the correctness of blocks and evaluate the terms. Concepts such as public keys, coinbase transactions, rewards, blocks, transaction fees are specified in Agda.

Acknowledgements

I would like to thank Dr Anton Setzer for guiding me with my thesis and providing me with great knowledge relevant to my study. I would also like to thank my family for inspiring me to have an interest in cryptography and stimulating my fascination in cryptocurrencies.

Table of Contents

Chapter 1

Introduction

Bitcoin and cryptocurrencies face severe regulations via governments worldwide. Research suggests many individuals fear cryptocurrencies and are mindful to foresee its benefits, due to the perplexing nature of the subject [1]. The ability to ban such technology is theoretically impossible hence it is crucial we understand digital currencies. Curiosity for cryptocurrencies is at a record high with numerous stakeholders having faith in decentralised solutions to current issues regarding inflation in fiat currencies caused by the pandemic. The concern around cryptocurrencies is the lack of knowledge, number of exploits, bad reputation due to infamously being linked to criminal activities, no central authority backing cryptocurrencies and the list goes on. However, we got to identify the advantages of cryptocurrencies. These cryptocurrencies have differing use cases. Bitcoin in particular acts to change the way people send / receive money essentially mimicking functionality of current fiat currency. Bitcoin is decentralized meaning users can send digital currency freely over the internet with no middle-man. I will be focusing on the concerns users have and plan to come up with ways to reduce ambiguity around the concept of Bitcoin and other cryptocurrencies. I will accomplish this by conducting proofs and verifying Bitcoin concepts. I have chosen Bitcoin as my main focus as it is the largest cryptocurrency as a result it has plenty of research already completed. However, Bitcoin still lacks in verification completed, there is only a few studies on Bitcoin verification and very limited verification carried out in my chosen language Agda. I will write manual proofs using Agda as its dependently typed functional language which will be useful to check correctness of Bitcoin.

1.1 Motivations

Fraudulent cryptocurrencies and exploitations with cryptocurrencies have been a prominent issue. This is largely due to the amount of traction cryptocurrencies has gained in recent times. As a result, cryptocurrencies are being heavily sanctioned and regulated due to the uncertainty surrounding cryptocurrencies. This thesis will present ways in which we can verify cryptocurrencies and demonstrate methods currently being applied to eliminate such problems which in turn strives to reduce number of regulations implemented. From this study developers of cryptocurrencies should be influenced to show importance to verification & specification stages. Programmers will recognize different techniques to verify concepts when creating cryptocurrencies. Following this study will be beneficial to have a greater understanding of concepts in Bitcoin and cryptocurrencies. The modelling of Bitcoin will also be illustrated which can be used as a reference when modelling other cryptocurrencies.

1.1.1 Objective

In this thesis I plan to achieve several objectives. Firstly, I will look at fundamental concepts within cryptocurrencies in depth which are relevant to understanding cryptocurrencies. This includes but not limited to the structure of Bitcoin, different cryptocurrency protocols, blockchain, proof of stake, cryptography, advantages & limitations of Bitcoin to name a few.

I will then look at Bitcoin and inform you on how bitcoin works and also give information on how to model Bitcoin. My key practical element of this thesis comes from programming where I expand on the specification of Bitcoin and show examples of proofs. I will accomplish this using Agda as my prover language. In the code I will provide corrective conditions, analyse current code, and clean up code for correctness. I will also parameterise the code as well as instantiate the code. It is also essential that I make sure the code is readable and I will accomplish this with the use of comments and following correct coding standards. I will use current code which is already created by Dr Anton Setzer and adapt as well as add to it when modelling Bitcoin [2]. I aim to create proofs and model the specification of Bitcoin concepts. There are so many concepts we can check for correctness however I shall be focusing on proving blockchain concept due to the complexity of proving several theorems in such timeframe as well as difficulty in preforming proofs manually, in the real world verification is commonly automated.

My intention behind this study is to show the necessity for cryptocurrencies to implement a formal specification and verification. I hope this study will demonstrate methods to have an effective verification stage.

1.2 Overview

I first outline the document structure shown in title of content and introduce what my thesis is about, the key contribution of this work is organized as follows.

I look at the concepts within cryptocurrencies as a whole providing you with crucial information to understanding cryptocurrencies. I will then look at Bitcoin specifically and explain concepts of Bitcoin that are relevant for modelling of Bitcoin

Then show examples of how to model and verify bitcoin concepts, by programming in Agda, I run proofs and model the Bitcoin specification.

Lastly I summarize the main contributions and key points to take away from conducting proofs as well as conclude the importance of verification and significance it has on cryptocurrencies.

1.3 Contributions

The main contributions of this work can be seen as follows:

- **Concepts within cryptocurrencies**

  I go over concepts within cryptocurrencies so that viewers have the relevant information on important concepts, how cryptocurrencies work and there use cases.

- **Agda**

  The programming language used and its advantages and disadvantages. I will also look at other common languages that are used in verification.

- **Modelling of Bitcoin**

  In this section I go into specifics on what I will be modelling and how I will model Bitcoin in Agda.

- **Specification and verification**

  This chapter will go in detail of the code written in Agda. It consists of the specification of Bitcoin concepts and examples of possible proofs. Modelling of Bitcoin concepts is my practical element of this study and can be viewed in the code listings section & appendix.

- **Final remarks**

  My results, importance of verification and future work in relation to my study.

Chapter 2

Cryptocurrencies concepts

Cryptocurrencies are digital currencies designed with several features such as to avoid the central authority, e.g., the government. Ecash was the first cryptographic technology invented in 1983. It all started when a cryptographer by the names of David Chaum released his dissertation on cryptography and privacy preserving technology. Ecash allowed users to maintain privacy across documents by the use of blind signatures [3]. Ecash was however a centralised system around DigiCash (corporation) and struggled to gain acceptance due to the number of patented algorithms it enforced which was frown upon from the crypto community [4]. In the 1990s other cryptographic ideas were proposed, implemented, and released as open source projects. Projects such as Hashcash, b-money and Bitgold all aided in the development of Bitcoin. Hashcash is a proof of work system used to limit email spam and denial of service attacks. Hashcash is known for its involvement in Bitcoin mining. B-money was another currency which intended to be an “anonymous distributed electronic cash system” however it was never implemented [4]. Bitgold was the earliest attempt at creating a decentralised virtual currency but was also never implemented. The main discovery from these cryptographic technologies was the use of cryptography and ideas presented in making currencies, leading up to the invention of Bitcoin which is known as the “first” cryptocurrency created [3]. Cryptographic technology has become so advanced today that many cryptocurrencies now provide more functionality in addition to being just a fiat currency alternative. Features such as smart contracts, synthetic tokens, non-fungible tokens, etc all perform tasks that one may never have imagined possible in the financial sector. One thing in common in most of these cryptocurrencies, is they still use technology concepts suggested decades prior [4].

2.1 Blockchain

Blockchain was popularised by Santoshi Nakamoto in 2008. Blockchain technology is a fundamental concept in most cryptocurrencies such as Bitcoin and Ethereum. Blockchain is a database which stores information digitally, its core difference between other databases is it stores data in groups known as blocks. These blocks have certain properties and storage capacities, they also link to other blocks therefore referred to as blockchain [5]. Bitcoin makes use of this public ledger to implement decentralised record of transactions. The key features of blockchain are that its immutable and as it can be a public ledger, it is also transparent [6] e.g. Bitcoin.

Figure 1.1 Simple blockchain

2.2 Proof of stake vs proof of work

Proof of stake and proof of work are mechanisms in cryptocurrencies to validate and allow the network to remain functional. Proof of work and proof of stake's aim is to reach an agreement between nodes as well as securing the network.

Proof of work algorithm, miners solve puzzle and are rewarded on based on first one to solve puzzle. Very electricity heavy. 54twh of electricity. Gives more rewards to higher hash rate nodes, mining pools combing hashing power evenly. (Blockchain is more centralized) [7].

Proof of stake uses an election process to select a random node to validate the block. Validators mint new blocks. To become a validator, you have to deposit certain number of coins as collateral. Size of stake determines chances of forging next block at a linear correlation. Fairer than proof of work. Validators check if all transactions in block is valid and signees it off as correct and adds it to the blockchain. If collateral stake is higher than reward, then unlikely for nodes to signing illegitimate blocks [8]

2.3 limitations of cryptocurrencies

- The main constraint of cryptocurrencies is the intrinsic value of the currency. If the public don't value cryptocurrencies the technology would be a redundant regardless of its advantages. This however is unlikely as cryptocurrencies market cap exceeds $2.13 trillion [9].

- Number of cryptocurrencies, this is an issue due to there being over 10,000 tokens in circulation [9]. As a result, we have tokens which are either fraudulent or copying the functionality of other tokens therefore making it impractical. This adds to the concerns the public observe when attempting to understanding cryptocurrencies.

- It's still a work in progress, there is so many stages and R & D when it comes to making cryptocurrencies. This means that cryptocurrencies are released as projects instead of as fully functional tokens, allowing for errors to be made during development stages of the cryptocurrency that may go undetected till

an exploitation arises. This is a problem even in large projects such as Ethereum.

- The number of electricity used by miners is an astonishing amount. Bitcoin is in the top 30 energy uses worldwide, using more energy than countries such as Argentina [9]. The total energy consumption has been a concerning issue for proof of work tokens. As it is a major concern for the environment and steps are being taken to divert from this issue however established tokens like Bitcoin still use proof of work mechanism.

- Dark web and Bitcoin. Bitcoins known for its uses as digital currency when users purchase goods illegally online. This is concerning due to the legal sector find this of great distress. This makes cryptocurrencies less secure as the legislations imposed on cryptocurrencies is forever in doubt.

2.4 Cryptography

Cryptography plays a key role in cryptocurrencies, after all its where it partially got its name from “crypto”. Cryptography is the technology behind securing cryptocurrencies by make sure transactions are processed automatically and securely. Cryptocurrencies uses public keys in particular to verify transactions [10]. A few cryptography concepts used are as follows: hash functions, asymmetric encryption, digital signature and Merkle trees. Further on in the paper I go into depth about such techniques used.

2.5 Other cryptocurrencies

Other cryptocurrencies which isn’t Bitcoin is referred to as alternative coins (alt coins) they differentiate themselves by having different use cases. Bitcoins use case is to act as a replacement to fiat currency and lead the revolution into the demand of digital currency. However, tokens such as Ethereum arguably has more functionality as it makes use of smart contracts which enable users to contractually agree to transactions online for virtually anything. Ethereum is the 2nd largest cryptocurrency. Cryptographers believe due to its functionality and active development, of which it aims at improving its sustainability and adding to its features it should be the biggest cryptocurrency. The 3rd biggest token by market cap is tether which is a stable coin that acts as the dollar, this is used for trading on exchanges. The 4th largest token is Binance coin, this is an exchange coin that allows investors to pay fees and trade by using (BNB) to purchase other tokens. As observed these tokens all have a different functionality and potentially more use cases than bitcoin.

Chapter 3

Programming Languages

3.1 Agda

Agda is a dependently typed programming language similar to Haskell. Agda is a proof assistant that allows proofs to be written in a functional programming style. Agda is a total language and therefore requires program to terminate [11].

3.1.1 Advantages of Agda

- Agda is similar to Haskell as it's a functional language and is built on Haskell. Agda is 55% Haskell and 27% Agda as shown in the git repository. Functional language is ideal when modelling cryptocurrencies and doing proofs as it's easy to read. Therefore, people of interest can follow the code with no real coding experience and still understand it with ease in comparison to object orientated language. [12]

- Agda also has great tools, emacs text editor provides syntax highlighting and is great for debugging. Agda also supports atom text editor.

- My favourite benefit from Agda is its type checking ability, Agda allows you to use holes in your code, with this you can write terms in holes. The main feature of the hole however is to make use of "Agda goal". You are able to case split, list context, infer type of expression, evaluate open terms and more [13].

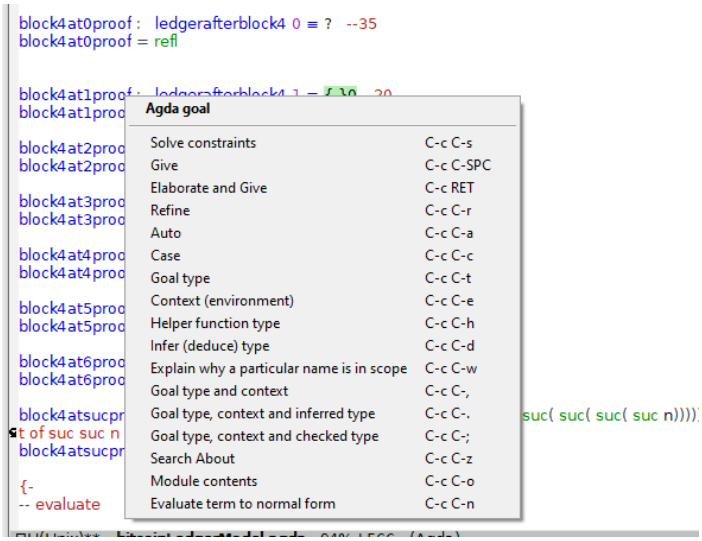

Figure 2.1: A demonstration of Agda type checking and Agda goals

- Agda has foreign function interface (FFI) allowing you to use Haskell libraries in agda and vice versa.

- Agda capability of metaprogrammming, although not relevant to my study.

- Termination checking, ability to check termination for each specific function. You can tag single functions and types to skip these checks. This is beneficial as when you run into an infinite loop you can easily search for functions that might be responsible.

3.1.2 Disadvantages of Agda

- The main disadvantage of Agda is that it is a small programming language only has 9 active developers and encourages the community to contribute fixes and create additional features [12]. This is problematic due to bugs are more likely to go undetected, it also means due to the limited developers time has to prioritised according to what is essential. In my opinion I believe that Agda could have prioritised time in making Agda installation go smoothly. Installing Agda and Agda library has been one of my worst experiences when it comes to installations. Installing Agda can only be installed via PowerShell, this is also a long-winded process, a solution to this would be to make Agda into a package that can be easily downloaded. Agda library is also hard to install due to the file being a tar file which requires special tool to be extracted.

Figure 2.2 Agda standard library installed as tar.gz file

- Agda in comparison to other languages such as Coq doesn't have as many libraries [13].

- Agda also has a several bugs. I have encountered a few myself however on GitHub you can see the list of bugs present awaiting fixes [12].

- Limited amount of information on learning Agda, there is only a handful of tutorials and articles in contrast to Coq.

- Agda is not fully finished language, it still needs time in development as some features are still not implemented.

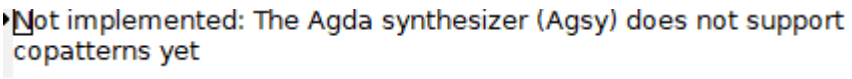

Figure 2.3 Feature not implemented.

3.2 Alternative Languages

A language which is related to Agda would be Coq which is an interactive theorem prover [14]. Coq is language which has greater user base and recognition. Coq was initially developed by Gérard Huet and Thierry Coquand as well as other researchers. Coq is now actively funded by INRIA [14]. In coq you can do higher order logic as well as dependently typed functional programming, its main focus however is theorem proving. Other languages also exist such as Idris and lean. I shall be modelling Bitcoin consisting of specification and verification therefore Agda is better suited for my project.

Chapter 4

Modelling of Bitcoin

In this chapter I will specifically look at Bitcoin concepts which are crucial to modelling Bitcoin. I will also delve into Bitcoin exploits.

4.1 Exploit and attacks

There have been several attacks and exploits against cryptocurrencies. Below are exploits and attacks against Bitcoin however there has been exploits at a larger scale in other cryptocurrencies such as Ethereum's DAO incident leading to a hard fork of Ethereum being implemented and resulting in the creation of Ethereum Classic.

4.1.1 legal attacks

This is when the government impose restrictions to the trading and possession of Bitcoin. Bitcoin has been banded several times by the government, for example China has banned cryptocurrencies by imposing banks from dealing with such currency.

4.1.2 Replay attacks

In the original bitcoin protocol uniqueness of transaction ID was not guaranteed, this allowed for replay attacks. Replay attacks is similar to a man in the middle attack but is more specific. An attacker intercepts then repeats a valid data transmission that goes through a network, the server will then send back a response to the attacker. This can be fixed by including the block number in Coinbase transaction [2].

4.1.3 Distributed Denial-Of-Service (DDOS) attack

This is when an attacker exhausts the network resources so that honest nodes don't get service or any information. This risk is mitigated in Bitcoin as its nodes are distributed across the world therefore making it financially infeasible [4].

4.1.4 Routing attack

A routing attack is when an attacker partitions the network into 2 or more disjoint components. This is achievable by preventing nodes within a component to interact with nodes outside therefore making parallel blockchains. All blocks mined in smaller component will be discarded with all the transactions and miner revenue. This attack still remains a threat in Bitcoin and measures can be taken to avoid this attack. The SABRE network can protect against this, its objective is to protect relay to relay and relay to client connections. It works by placing nodes in ISPs that peer to each other. This forms a densely connected graph of direct peering links. To protect relay to client it places relay nodes so that clients have at least one path to SABRE [15].

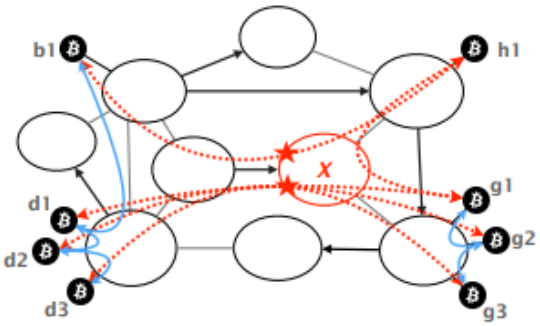

Figure 3.1 Source:[15] Network split into partitions.

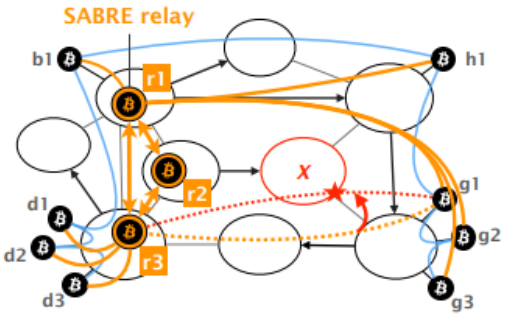

Figure 3.2 Source:[15] Implementation of SABRE, resulting in network still being connected.

4.1.5 Eclipse attack

The attacker seeks to isolate and attack single nodes of the network. By isolating targeted node attacker can produce illegitimate transaction confirmations as a result attacker is able to disrupt other network nodes. This attack can be mitigated with the use of random node selection.

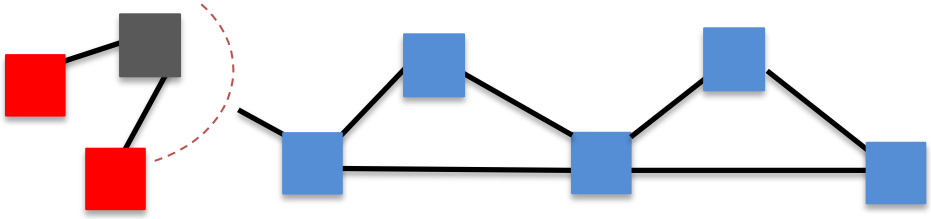

Figure 3.3 Image of a network which has been affected by eclipse attack. As shown node is isolated from network.

4.1.6 51% attack

The 51% is an attack when a group of attackers control more than 50% of the blockchain network. In such instance the consensus of the network is no longer viable thus resulting in the blockchain to be manipulated. Attackers can reverse, duplicate, and stop transactions which will be detrimental [16]. This attack has become more likely due to the use of mining pools. Mining pools allow miners to combine their computational resource to mine blocks and share the reward. This has become very useful as mining blocks require great amount of power. However, it is a concern as these pools can control more than 50%. An example of this was when GHash.io pool briefly gained 50%+ of Bitcoin network's computing power in 2014 [16]. Thankfully nothing was exploited but it demonstrates the threat of this attack in relation current mining techniques. 51% attack was considered as a hypothetical attack up till mining pools as it was presumed infeasible for such attack to occur.

4.1.7 Double spending exploit

Refers to the ability for an individual to spend the same balance more than once on a transaction. For example, a user could have 1BTC and send 1BTC to 2 people if both transactions are pulled from the pool for confirmation simultaneously. Satoshi Nakamoto offers a solution to this exploit by using a timestamped server to generate computational proof of the chronological order of transactions. This solution is only compromised when fraudulent nodes control more CPU power than honest nodes in the system [8].

4.2 Forks

A fork occurs when the blockchain splits into 2 blockchains. There are 2 types of forks, hard fork, and soft fork. Soft forks are backward compatible allowing nodes which accepted the update to interact with nodes that may not have. The hard fork however isn't backward compatible as the update includes changes that significantly alter the original blockchain [4]. Hard forks are commonly used to correct security concerns from previous blockchain. In the Ethereum blockchain this was demonstrated when they created a hard fork to reverse the exploit on the DAO. In relation to Bitcoin forking has occurred to create other projects that are fundamentally similar but have a few amendments. Bitcoin Cash is a cryptocurrency which is a hard fork of Bitcoin [4].

4.3 Bitcoin Ledger

Bitcoin ledger is essentially the way in which records of transactions between addresses are stored digitally. It's stored as a ledger. Bitcoin does this by using blockchain technology and each block contains specific data needed for the Bitcoin ledger to function. In this block you have data such as transaction ID, amount of bitcoin, sender address, receipt address, time stamp and hash of block. Arguably

blockchain is the most important concept in cryptocurrency and is used universally. I will be modelling the blockchain but first we have to understand blockchain.

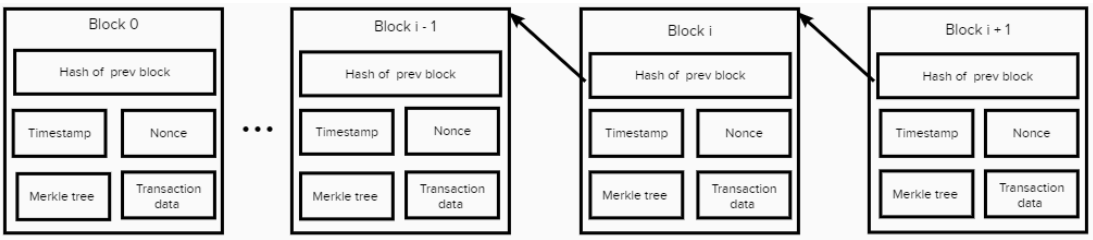

Figure 3.4 Example model of blockchain

4.3.1 Transactions

There are 2 types of transactions. Coinbase and regular transactions. Coinbase transactions are used for new bitcoins contrary to regular which is used for transferring existing bitcoin from one user to another [17]. Transactions consist of input and outputs. The input transaction is the output of the previous transaction (indicates funds spent from previous transaction). The output transaction is the amount of bitcoin transferred to address. It consists of the amount and a challenge which specifies the condition under which the transaction can be claimed. Coinbase transactions have no input and grant reward to the miner who added the block [18].

4.4 Cryptographic hash function

A hash function is a function that takes in an input and maps it to a fixed size output, known as the hash. Hash functions are one-way functions and cannot be reversed to reveal the input. Hash functions have various purposes in the blockchain, for example, solving cryptographic puzzles, address generation, signature generation and verification [4]. Bitcoin uses SHA-256 algorithm which is a hashing algorithm used to validate its transactions. The only method in getting the original input is to use Brute force search on all the inputs and produce outputs from the inputs and check if it matches with said hash. This would take millions of years to crack making it impractical [4]. Hash functions have the following properties: deterministic, pre-image resistant and. collision resistant

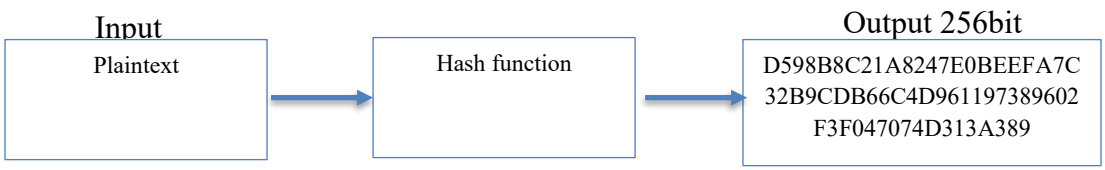

Figure 3.5: Simple hash function

4.4.1 Merkle tree

Otherwise known as hash tree, is a data structure of which each node is a hash of a data blocks. Merkle trees are utilized in the blockchain to secure verification of information. The Merkle tree consists of the root (Merkle root) which is found at the top of the tree. The Merkle root of transactions is stored in the blockchain [4].

4.5 Digital keys and signatures

Keys are pronominally known from cryptography. Asymmetric encryption is used for secure communication, it uses a public key for encryption and a private key for decryption. However, Bitcoin does not use this technology, as encryption is not implemented in communication and transaction data. Bitcoin makes use of digital keys which are created by users and stored in wallets. Keys come in pairs which consist of private key and public key. Public key can be referred to as the account details and the private key provides users control over the account [19]. When making a transaction you need the recipients public key, this is the Bitcoin address. Signatures can only be created with private keys. Signatures are referred to as "witness" as it validates the ownership of funds being spent [19]. Essentially asymmetric encryption is for the signature process on every transaction. This also ensuring only the owner of private key can produce signatures.

4.6 Mining

Mining in Bitcoin is needed to find consensus for if it's a correct block. Mining is the process of solving puzzles in a proof of work system to verify transactions and add new blocks to the blockchain. Nodes which mine are compensated for partaking in verifying transactions with rewards which is typically the digital currency. Mining is resource intensive process which requires huge amount of computational power and memory [2]. This is why mining pools exist which combine power to solve such puzzles. Every transaction on the blockchain has transaction fees this is the fee which goes to the miner who mines the block including your transaction [17]. In mining there is also concept called maturation time. This is in relation to coinbase transactions, miners are required to mine 100 blocks before coinbase transactions can be used [2]. This is executed as sometimes blockchain fork, making blocks that were valid, invalid. The miner reward of those blocks is lost, if miners were able to spend the reward and the block got orphaned then the recipients would have their bitcoin disappear. A fork with 100 blocks is less likely to experience such issue due to the greater number of blocks [20].

Proof of stake systems rely on validators who stake their digital currency so that blocks can be verified. They are also rewarded similarly to proof of work. As discussed previously validators are more environmentally friendly as significantly less computational power is required [8].

Chapter 5

Specification and verification

In this chapter we will look at specifying bitcoin concepts as well as verifying correctness of the specification.

5.1 Specification of Bitcoin

In order to verify bitcoin concepts, we need to model the specification of Bitcoin. I will be adding to code currently available on the Bitcoin ledger model [2]. This models bitcoin blockchain. I will be focusing on modelling individual blocks of the blockchain and conducting runs with examples. As I have used Agda as my programming language of choice its necessary to go over basic elements of the language and how to use Agda. This is beneficial as Agda is a fairly small language therefore some terms may be difficult to understand.

5.2 Introduction to Agda

Dependent types- Agda is known as a dependently typed language. Dependent types are types which depend on a certain value. The expression shown in figure 4.1 shows that Msg depends on argument n. Figure 4.2 is another example in which the parentheses, variable and: signal a dependent type [21].

```
(n : ℕ)        → Msg
```

Figure 4.1: Expression of dependent function

```
module test  where

test :  (A B : Set) → A → B → A
test = λ A B a b → a
```

Figure 4.2: Dependent types

Data types- Types can be defined and there also exists built in data types such as Agda.Builtin.Equality, Agda.Builtin.Nat and Agda.Builtin.Bool. It is fairly simple however to create data types in Agda.

```
--simple data type bools

data Bool : Set where
 true : Bool
 false : Bool     -- closely to haskell, set type for bool data type, types have type
```

Figure 4.3: Data type Boolean

Implicit argument

```
-- define if
if_then_else : {A : Set}→ Bool → A → A → A    -- A:set is used to define a as set
if true then x else y = x
if false then x else y = y  -- used case split
```

4.4 Data type if-else condition

Implicit arguments- Are arguments with "{}". They can be omitted as long as they can be inferred by the type checker [2].

Parameterised and indexed datatypes- Parameterised datatypes are used to create a family of related datatypes. They are declared after the name of the datatype and

before the colon [22]. Indexed datatypes are generalisation of parameterised datatypes. Parameters are required to be the same for all constructors, indices can vary from constructor to constructor [22]. In parameterised datatypes all constructor result types are the same, whereas in indexed we need more expressivity [21]. Example below.

```
data ListI : Set → Set₁ where
  zero  : {A : Set} → ListI A
  _::_ : {A : Set} → A → ListI A → ListI A
```

Figure 4.5: Parameterised

```
data List (A : Set) : Set where
  [] : List A
  cons  : A → List A → List A
```

4.6 Indexed, extra expressions

Record types- Allow you to put values together, values are tuples of said type [21]. In the example below we have record type TXField which introduces record type amount and address.

```
record TXField : Set where
  constructor txField
  field  amount    : Amount
         address   : Address
```

Figure 4.7: Record type

Logic- In Agda you can also define propositional and predicate logic. This comes in very handy when defining new datatypes. Useful for when you do proofs with logic [21].

```
data _∧_ (A B : Set) : Set where
  ∧-introduction : A → B → A ∧ B
```

Figure 4.8 Conjunction definition

```
data _∨_ (A B : Set) : Set where -- _x_ notation
  ∨-intro₀ : A → A ∨ B -- disjuntion
  ∨-intro₁ : B → A ∨ B
```

Figure 4.9 Disjunction definition

Equality- This checks for relation between variables. In Agda it is denoted by ≡ and has as a constructor a proof of reflexivity [2].

Postulate- Postulate allow you to introduce elements in a type without defining the element. E.g. `postulate close : False.` However, postulate should be used with caution as one could introduce an element in the empty set resulting in inconsistent definitions [22].

Totality- is when the functions terminate successfully, this is requirement in Agda all functions must terminate. This feature is important as we are able to know the programs will always evaluate successfully. Agda is able to validate totality by the use of type checking, coverage checking and termination checking [22].

Type checking allows you to check for run time errors, coverage checking allows you to check for partial functions and termination checking will check for structural recursion.

5.3 Specification of Bitcoin

Firstly, we have to define the basic data types: address (bitcoin address), amount (number of bitcoins spent/received), TXID (transaction id), PublicKey and Signature (digital signature). All these types are natural numbers ℕ e.g. TXID = ℕ
Next we define data types, messages with hash functions to map to natural numbers. Messages can be a natural number, or a list of messages. Using messages defined, we can create numerous instances of messages such as list of input/output of a transaction [2].

```
data Msg : Set where
  nat    : (n : ℕ)          → Msg
  _+msg_ : (m m' : Msg)     → Msg
  list   : (l : List Msg)   → Msg

postulate hashMsg : Msg → ℕ
```

We then create a function for public key to address as explained previously this address is generated with public key. We also created a postulate function so that unimplemented addresses (n) are considered.

```
postulate publicKey2AddressUnimplemented : (pubk : PublicKey) → Address
publicKey2Address : (pubk : PublicKey) → Address
publicKey2Address 4 = 3 --ADDED
publicKey2Address n = publicKey2AddressUnimplemented n -- unkown n
```

Signed is now defined, this is so that we can check whether a message is signed by the corresponding public key. Signed represents the different block transactions which results to true (⊤). Similarly, to above we also postulate signed as for some values signed is not implemented.

```
postulate SignedUnimplemented : (msg : Msg)(publicKey : PublicKey)(s : Signature) → Set


Signed : (msg : Msg)(publicKey : PublicKey)(s : Signature) → Set
Signed ((nat 5 +msg nat 0) +msg list ((nat 5 +msg nat 1) :: [])) 17 15 = ⊤ -- 15 is the signature
Signed ((nat 10 +msg nat 0) +msg   list ((nat 12 +msg nat 2) :: (nat 5 +msg nat 3) :: [])) 17 8 = ⊤
                                   -- (signature2)
 Signed msg publicKey s = SignedUnimplemented  msg publicKey s
```

We introduce a set SignedWithSigPbk which has the following: public key, signature, status of if message was signed for and if address is the hash of the public key. SignedWithSigPbk is a record type for, which consists of parameters msg, address and signature [2].

```
record SignedWithSigPbk (msg : Msg)(address : Address) : Set where
 field publicKey  :  PublicKey
       pbkCorrect : publicKey2Address publicKey ≡ℕ  address
       signature  :  Signature
       signed     :  Signed msg publicKey signature


open SignedWithSigPbk public
```

Now we specify transactions. The transactions are calculated by input (amount received) and the output (amount sent) in relation to the address. We can model this with record types.

```
record TXField : Set where
  constructor txField
  field  amount    : Amount
         address   : Address

open TXField public
```

A message is created for the transaction field, mapL is the function that applies a function to each element in a list [2].

```
txField2Msg : (inp : TXField) → Msg
txField2Msg inp  =   nat (amount inp) +msg nat (address inp)

txFieldList2Msg : (inp : List TXField) → Msg
txFieldList2Msg inp  = list (mapL txField2Msg inp)
```

Function to calculate all the input and outputs is computed.

```
txFieldList2TotalAmount : (inp : List TXField) → Amount
txFieldList2TotalAmount inp = sumListViaf amount inp
```

Modelling for unsigned transactions are also needed.

```
record TXUnsigned : Set where
    inputs   : List TXField
    outputs : List TXField
```

We calculate a message for this unsigned transaction.

```
txUnsigned2Msg : (transac : TXUnsigned) → Msg
txUnsigned2Msg transac = txFieldList2Msg (inputs transac) +msg txFieldList2Msg (outputs transac)
```

For an input the message to be signed consists of the input and the list of outputs [2].

```
txInput2Msg : (inp : TXField)(outp : List TXField) → Msg
txInput2Msg inp outp = txField2Msg inp +msg txFieldList2Msg outp
```

The set of signatures for transactions have signatures for each input using the txInput2Msg function [2].

```
tx2Signaux : (inp : List TXField) (outp : List TXField)  → Set
tx2Signaux []           outp  = ⊤
tx2Signaux (inp :: restinp)  outp  =
   SignedWithSigPbk (txInput2Msg inp outp) (address inp) ×  tx2Signaux restinp outp

tx2Sign : TXUnsigned → Set
tx2Sign tr = tx2Signaux (inputs tr) (outputs tr)
```

We also model a transaction which is an unsigned transaction, sum of outputs is ≥ the sum of inputs. The list of input and output is non-empty which is also signed by the private key of the public key, this hashes to the input address [2].

```
record TX : Set where
  field  tx      :  TXUnsigned
         cor     : txFieldList2TotalAmount (outputs tx) ≦ txFieldList2TotalAmount (inputs tx)
         nonEmpt : NonNil (inputs tx) × NonNil (outputs tx)
         sig     : tx2Sign tx

open TX public
```

The ledger functions for every time and address, amount is correlated. Thus, meaning a ledger is a function from address to amount. The empty ledger is also stated with amount = 0 [2].

```
Ledger : Set
Ledger = (addr : Address) → Amount
initialLedger : Ledger
initialLedger addr = 0
```

We have to now define functions when the ledger is updated (when transactions are added). This operation is extended to a list of transactions: [2]

```
addTXFieldToLedger :  (tr : TXField)(oldLedger : Ledger) → Ledger
addTXFieldToLedger  tr oldLedger pubk =
       if   pubk ≡ℕb address tr then oldLedger pubk +  amount tr else oldLedger pubk

addTXFieldListToLedger  :  (tr : List TXField)(oldLedger : Ledger) → Ledger
addTXFieldListToLedger []       oldLedger  =  oldLedger
addTXFieldListToLedger (x :: tr)  oldLedger  =
    addTXFieldListToLedger tr (addTXFieldToLedger x oldLedger)
```

Likewise, this is also done for the subtracting a ledger field. It follows the same structure as addition ledger but the + is replaced with -. We also have to make sure amount doesn't go into a negative value we achieve this with the use of "∸" which cuts off at 0 [2].

```
subtrTXFieldFromLedger     : (tr : TXField)      (oldLedger : Ledger)  →  Ledger
subtrTXFieldListFromLedger : (tr : List TXField) (oldLedger : Ledger)  →  Ledger
subtrTXFieldFromLedger  tr oldLedger pubk =
      if   pubk ≡ℕb address tr  then oldLedger pubk ∸  amount tr  else oldLedger pubk

subtrTXFieldListFromLedger [] oldLedger = oldLedger
subtrTXFieldListFromLedger (x :: tr) oldLedger =
       subtrTXFieldListFromLedger tr (subtrTXFieldFromLedger x oldLedger)
```

We also have to include the updated ledger; it is updated by subtracting the input then adding the output.

```
updateLedgerByTX :  (tr : TX)(oldLedger : Ledger) → Ledger
updateLedgerByTX tr oldLedger =  addTXFieldListToLedger (outputs (tx tr))
                                 (subtrTXFieldListFromLedger (inputs (tx tr)) oldLedger )
```

This function is to check correct input of ledger. It is valid if there is sufficient amount in the ledger to process transaction [2].

```
correctInput : (tr : TXField) (ledger : Ledger) → Set
correctInput tr ledger = amount tr ≦ ledger (address tr)
```

The list of input in ledger is correct if each input is correct, whilst updating the ledger after every input. This function is extended to transactions [2].

```
correctInputs : (tr : List TXField) (ledger : Ledger) → Set
correctInputs []       ledger  = ⊤
correctInputs (x :: tr)  ledger  = correctInput x ledger ×
                        correctInputs tr (subtrTXFieldFromLedger x ledger)

correctTX : (tr : TX) (ledger : Ledger) → Set
correctTX tr ledger = correctInputs (inputs (tx tr)) ledger
```

An unmined block requires no correctness checks and is a list of transactions [2].

```
UnMinedBlock : Set
UnMinedBlock = List TX
```

We now model the transaction fee, which is the difference between the sum of inputs and sum of outputs of the transaction. For unmined block this differs, it is the sum of transaction fees of its transaction [2].

```
tx2TXFee : TX → Amount
tx2TXFee tr =
  txFieldList2TotalAmount (outputs (tx tr)) - txFieldList2TotalAmount (inputs (tx tr))

unMinedBlock2TXFee : UnMinedBlock → Amount
unMinedBlock2TXFee bl = sumListViaf tx2TXFee  bl
```

A block is correct if the transaction is correct whilst updating the ledger after each transaction. The output of updated ledger after block is also computed.

```
correctUnminedBlock : (block : UnMinedBlock)(oldLedger : Ledger)→ Set
correctUnminedBlock  []          oldLedger  = ⊤
correctUnminedBlock  (tr :: block)  oldLedger  =
  correctTX tr oldLedger ×  correctUnminedBlock  block (updateLedgerByTX tr oldLedger)

updateLedgerUnminedBlock : (block : UnMinedBlock)(oldLedger : Ledger) → Ledger
updateLedgerUnminedBlock []          oldLedger  = oldLedger
updateLedgerUnminedBlock (tr :: block)  oldLedger  =
  updateLedgerUnminedBlock block (updateLedgerByTX tr oldLedger)
```

The BlockUnchecked function is an unmined block with the output given to the miner. The miner adds 1 transaction to the block (coinbase transaction), this transaction has no inputs. The sum of outputs is therefore equal to the block reward and transaction fee of block.

```
BlockUnchecked : Set
BlockUnchecked = List TXField × UnMinedBlock

block2TXFee : BlockUnchecked → Amount
block2TXFee (outputMiner , block) = unMinedBlock2TXFee block
```

A block is correct if the unmined block is correct and the output of coinbase transaction is equal to the sum of the block reward and transaction fees [2].

```
correctMinedBlock : (reward : Amount)(block : BlockUnchecked)(oldLedger : Ledger) → Set

correctMinedBlock reward (outputMiner , block) oldLedger =
    correctUnminedBlock block oldLedger ×
    txFieldList2TotalAmount outputMiner ≡ℕ (reward + unMinedBlock2TXFee block)
```

The ledger is updated after a block. We update the ledger by using the unmined block and the output of coinbase transaction.

```
updateLedgerBlock : (block : BlockUnchecked)(oldLedger : Ledger)→ Ledger
updateLedgerBlock (outputMiner , block) oldLedger =
  addTXFieldListToLedger  outputMiner (updateLedgerUnminedBlock block oldLedger)
```

Blockchain unchecked is a list of unchecked blocks.

```
BlockChainUnchecked : Set
BlockChainUnchecked = List BlockUnchecked
```

The correctness of blockchain depends on the block reward. Block reward depends on the time in this case the number of blocks. The correctness can be calculated assuming the start time and start ledger is = 0 [2].

```
CorrectBlockChain : (blockReward : Time → Amount)
                    (startTime : Time)
                    (sartLedger : Ledger)
                    (bc  : BlockChainUnchecked)
                    → Set
CorrectBlockChain blockReward startTime startLedger [] = ⊤
CorrectBlockChain blockReward startTime startLedger (block :: restbc)
  = correctMinedBlock (blockReward startTime) block startLedger
    ×  CorrectBlockChain blockReward (suc startTime)
      (updateLedgerBlock block startLedger)  restbc
```

The final ledger is the end of the blockchain. It depends on the transaction fee and block reward and is calculated using the start time and start leger.

```
FinalLedger :  (txFeePrevious : Amount)     (blockReward : Time → Amount)
               (startTime : Time)          (startLedger : Ledger)
               (bc  : BlockChainUnchecked)  → Ledger
FinalLedger trfee blockReward startTime startLedger [] = startLedger
FinalLedger trfee blockReward startTime startLedger (block :: restbc) =
  FinalLedger (block2TXFee block) blockReward (suc startTime)
    (updateLedgerBlock block startLedger) restbc
```

Blockchain is a record type which introduces 2 projection functions (blockchain and correct). A blockchain is an unchecked blockchain [2].

```
record BlockChain (blockReward : Time → Amount) : Set where
  field  blockchain  : BlockChainUnchecked
         correct     : CorrectBlockChain blockReward 0 initialLedger blockchain
```

We also have a function for the ledger at the end of a blockchain.

```
blockChain2FinalLedger :  (blockReward : Time → Amount)(bc : BlockChain blockReward)
                       → Ledger
blockChain2FinalLedger blockReward bc =
  FinalLedger 0 blockReward 0 initialLedger (blockchain bc)
```

To model blocks specifically we need to first define the block reward. Reward is calculated by time spent.

```
myReward : Time → Amount
myReward zero = 50
myReward (suc zero) = 25
myReward (suc (suc zero)) = 12
myReward (suc (suc (suc t))) = 6
```

We can now model the empty block which is defined below.

```
bcemptyunchecked : BlockChainUnchecked
bcemptyunchecked = []


correctbcemptyunchecked : CorrectBlockChain myReward 0 initialLedger bcemptyunchecked
correctbcemptyunchecked = tt

bcempty : BlockChain myReward
blockchain bcempty = bcemptyunchecked
correct bcempty = correctbcemptyunchecked
```

The ledger is now updated so that we can observe the ledger after empty blockchain.

```
ledgerafterempty : Ledger
ledgerafterempty = blockChain2FinalLedger myReward bcempty
```

The block with 1 element is now modelled. For this we have input of amount 50 and no output at address 0. The value is 50 due to that's the value set as reward.

```
block1 : BlockUnchecked
block1 .proj₁ = txField 50 0 :: []
block1 .proj₂ = []

bc1elem : BlockChainUnchecked
bc1elem = block1 :: []

bc1 : BlockChain myReward
blockchain bc1 = bc1elem
correct bc1 = (tt , tt) , tt
```

Again, we now update the ledger to include block 1.

```
ledgerafterblock1 : Ledger
ledgerafterblock1 = blockChain2FinalLedger myReward bc1
```

Now we have to create a transaction for the following blocks as we now include output for block 2 as well, this is modelled below. We have input of 5 from address 0 which is send to address 1 as output. The variables check for numerous conditions. cor checks if the sum of input is greater than the sum of output.
tx2 .nonEmpt .proj₁ = tt and tx2 .nonEmpt .proj₂ = tt checks against true if the input & output list is non- empty. We then define the public key as a random integer, theoretically this would be a hashed value. The signature is also assigned, and we check for other true conditions (correct public key and signature).

```
tx2Unsigned : TXUnsigned
inputs tx2Unsigned = txField 5 0 :: []
outputs tx2Unsigned = txField 5 1 :: []
tx2 : TX
tx tx2 = tx2Unsigned
cor tx2 = tt
tx2 .nonEmpt .proj₁ = tt
tx2 .nonEmpt .proj₂ = tt
publicKey (proj₁ (sig tx2)) = 17
pbkCorrect (proj₁ (sig tx2)) = tt
signature (proj₁ (sig tx2)) = 15
signed (proj₁ (sig tx2)) = tt
proj₂ (sig tx2) = tt
```

In accordance with the reward function, we give the block a value of 25.

```
block2 :  BlockUnchecked
proj₁ block2 = txField 25 1 :: []
proj₂ block2 = tx2 :: []


bc2elem : BlockChainUnchecked
bc2elem = block1 :: block2 :: []

bc2 : BlockChain myReward
blockchain bc2 = bc2elem
correct bc2 = (tt , tt) , ((((tt , tt) , tt) , tt) , tt)
ledgerafterblock2 : Ledger
ledgerafterblock2 = blockChain2FinalLedger myReward bc2
```

The process is repeated for block 3 however we have 2 inputs, and 2 outputs therefore require more checks for transaction.

```
tx3Unsigned : TXUnsigned
inputs tx3Unsigned = txField 10 0 :: txField 7 1 :: []
outputs tx3Unsigned = txField 12 2 :: txField 5 3 :: []


tx3 : TX
tx tx3 = tx3Unsigned
cor tx3 = tt
proj₁ (nonEmpt tx3) = tt
proj₂ (nonEmpt tx3) = tt
publicKey (proj₁ (sig tx3)) = 17
pbkCorrect (proj₁ (sig tx3)) = tt
signature (proj₁ (sig tx3)) = 8
signed (proj₁ (sig tx3)) = tt
publicKey (proj₁ (proj₂ (sig tx3))) = 5

pbkCorrect (proj₁ (proj₂ (sig tx3))) = tt
signature (proj₁ (proj₂ (sig tx3))) = 9
signed (proj₁ (proj₂ (sig tx3))) = tt
proj₂ (proj₂ (sig tx3)) = tt


block3 :  BlockUnchecked
proj₁ block3 = txField 12 2 :: []
proj₂ block3 = tx3 :: []


bc3elem : BlockChainUnchecked
bc3elem = block1 ::  block2 :: block3 ::  []


bc3 : BlockChain myReward
blockchain bc3 = bc3elem
correct bc3 = (tt , tt) , ((((tt , tt) , tt) , tt) , ((((tt , (tt , tt)) , tt) , tt) , tt))

ledgerafterblock3 : Ledger
ledgerafterblock3 = blockChain2FinalLedger myReward bc3
```

Block 4 has 3 inputs and 3 outputs, its similar to above with the same checks just different signatures and keys. The number of checks is greater due to the number of inputs, outputs, public keys, and signatures.

```
tx4Unsigned : TXUnsigned
inputs tx4Unsigned = txField 3 1 :: txField 8 2 :: txField 4 3 :: []
outputs tx4Unsigned = txField 5 4 :: txField 2 5 :: txField 8 6  :: []
```

```
tx4 : TX
tx tx4 = tx4Unsigned
cor tx4 = tt
proj₁ (nonEmpt tx4) = tt
proj₂ (nonEmpt tx4) = tt
publicKey (proj₁ (sig tx4)) = 5
pbkCorrect (proj₁ (sig tx4)) = tt
signature (proj₁ (sig tx4)) = 6
signed (proj₁ (sig tx4)) = tt
publicKey (proj₁ (proj₂ (sig tx4))) = 3
pbkCorrect (proj₁ (proj₂ (sig tx4))) = tt
signature (proj₁ (proj₂ (sig tx4))) = 4
signed (proj₁ (proj₂ (sig tx4))) = tt
publicKey (proj₁ (proj₂ (proj₂ (sig tx4)))) = 4
pbkCorrect (proj₁ (proj₂ (proj₂ (sig tx4)))) = tt
signature (proj₁ (proj₂ (proj₂ (sig tx4)))) = 5
signed (proj₁ (proj₂ (proj₂ (sig tx4)))) = tt
proj₂ (proj₂ (proj₂ (sig tx4))) = tt


block4 :  BlockUnchecked
proj₁ block4 = txField 3 3 :: txField 3 4 ::  []
proj₂ block4 = tx4 :: []


bc4elem : BlockChainUnchecked
bc4elem = block1 ::  block2 :: block3 :: block4 :: []

bc4 : BlockChain myReward
blockchain bc4 = bc4elem
correct bc4 = (tt , tt) , ((((tt , tt) , tt) , tt) , ((((tt , (tt , tt)) , tt) , tt) , ((((tt , (tt , (tt , tt))) , tt) , tt) , tt)))

ledgerafterblock4 : Ledger
ledgerafterblock4 = blockChain2FinalLedger myReward bc4
```

5.4 Verification of Bitcoin

We verify the individual blocks in the blockchain. Block_at_proof carries at specific proofs to check the value outputted is correct. It does this by checking for equality and termination confirms correctness of the modelling of the blocks. We can identify with the proof the value returned at each address. Running actual examples allowed us to verify the specification of the ledger model. If we evaluate term to normal form and run ledgerafterblock_ x then we can check the process step by step. This is highlighted in red below.

```
block1at0proof :   ledgerafterblock1 0 ≡ 50 -- 50! verfication provided
block1at0proof = refl
block1atsucproof : (n : ℕ) → ledgerafterblock1 (suc n) ≡ 0
block1atsucproof n = refl

evaluation ledgerafterblock1 n

if n ≡ℕb 0 then 50 else 0

ledgerafterblock1 (suc n)

0
```

```
block2at0proof :   ledgerafterblock2 0 ≡ 45 -- 45!verfication provided
block2at0proof = refl

block2at1proof :   ledgerafterblock2 1 ≡ 30
block2at1proof = refl

block2atsucproof : (n : ℕ) → ledgerafterblock2 (suc( suc n)) ≡ 0
block2atsucproof n = refl
result of evaluting ledgerafterblock2 n

if n ≡ℕb 1 then
(if n ≡ℕb 1 then
 (if n ≡ℕb 0 then (if n ≡ℕb 0 then 50 else 0) ∸ 5 else
  (if n ≡ℕb 0 then 50 else 0))
 + 5
 else
 (if n ≡ℕb 0 then (if n ≡ℕb 0 then 50 else 0) ∸ 5 else
  (if n ≡ℕb 0 then 50 else 0)))
+ 25
else
(if n ≡ℕb 1 then
 (if n ≡ℕb 0 then (if n ≡ℕb 0 then 50 else 0) ∸ 5 else
  (if n ≡ℕb 0 then 50 else 0))
 + 5
 else
 (if n ≡ℕb 0 then (if n ≡ℕb 0 then 50 else 0) ∸ 5 else
  (if n ≡ℕb 0 then 50 else 0)))

result of evaluating result after  ledgerafterblock2 (suc( suc n))

0

result of evaluating result after  ledgerafterblock2 (suc n)

if n ≡ℕb 0 then (if n ≡ℕb 0 then 5 else 0) + 25 else
(if n ≡ℕb 0 then 5 else 0)


block3at0proof :   ledgerafterblock3 0 ≡ 35 -- 35!verfication provided
block3at0proof = refl

block3at1proof :   ledgerafterblock3 1 ≡ 23
block3at1proof = refl

block3at2proof :   ledgerafterblock3 2 ≡ 24
block3at2proof = refl

block3at3proof :   ledgerafterblock3 3 ≡ 5
block3at3proof = refl

block3atsucproof : (n : ℕ) → ledgerafterblock3 (suc( suc( suc( suc n)))) ≡ 0
block3atsucproof n = refl
```

```
ledgerafterblock3 n

if n ≡ℕb 2 then
(if n ≡ℕb 3 then
 (if n ≡ℕb 2 then
  (if n ≡ℕb 1 then
   (if n ≡ℕb 0 then
    (if n ≡ℕb 1 then
     (if n ≡ℕb 1 then
      (if n ≡ℕb 0 then (if n ≡ℕb 0 then 50 else 0) ∸ 5 else
       (if n ≡ℕb 0 then 50 else 0))
      + 5
      else
... 600+ lines of statements

ledgerafterblock3 (suc( suc( suc( suc n))))

0



block4at0proof :  ledgerafterblock4 0 ≡ 35  --35
block4at0proof = refl


block4at1proof :  ledgerafterblock4 1 ≡ 20 --20
block4at1proof = refl

block4at2proof :  ledgerafterblock4 2 ≡ 16
block4at2proof = refl

block4at3proof :  ledgerafterblock4 3 ≡ 4
block4at3proof = refl

block4at4proof :  ledgerafterblock4 4 ≡ 8
block4at4proof = refl

block4at5proof :  ledgerafterblock4 5 ≡ 2
block4at5proof = refl

block4at6proof :  ledgerafterblock4 6 ≡ 8
block4at6proof = refl

block4atsucproof : (n : ℕ) → ledgerafterblock4 (suc( suc( suc( suc (suc( suc( suc( suc n)))))))) ≡ 0
block4atsucproof n = refl
```

```
-- evaluate

ledgerafterblock4 (suc( suc( suc( suc (suc( suc( suc( suc n))))))))

0

ledgerafterblock4 n

if n ≡ℕb 2 then
(if n ≡ℕb 1 then
 (if n ≡ℕb 4 then
  (if n ≡ℕb 3 then
   (if n ≡ℕb 2 then
    (if n ≡ℕb 1 then
     (if n ≡ℕb 0 then
      (if n ≡ℕb 0 then
       (if n ≡ℕb 1 then (if n ≡ℕb 0 then 12 else 0) + 5 else
        (if n ≡ℕb 0 then 12 else 0))
       + 12
       else
..... 200lines of statements
```

5.6 Bugs

The first substantial bug was found in the library and resulted in falsity for a true condition.

```
nonNil : {X : Set}(l : List X) → Bool
nonNil [] = true   -- bug in code leads to falsity
nonNil (_ :: _) = false
```

Opposed to

```
nonNil : {X : Set}(l : List X) → Bool
nonNil [] = false
nonNil (_ :: _) = true
```

Led to error message

```
⊤ !=< ⊥ of type Set
when checking that the expression tt has type
NonNil (inputs (tx tx2))
```

Another bug was found in the code when modelling correctTX which also led to falsity.

```
correctTX : (tr : TX) (ledger : Ledger) → Set
correctTX tr ledger = correctInputs (inputs (tx tr)) ledger   -- was wrong
                                    -- was (outputs (tx tr))
```

Compatibility issues- older versions of Agda didn't allow private definitions, this was puzzling as newer versions allow private definitions. This resulted in difficult to detect bugs.

Chapter 6

Conclusions and Future Work

In this dissertation I have gone through vital cryptocurrency concepts as well as concepts needed to model Bitcoin. The main practical element of this study was to conduct correctness conditions, modelling bitcoin specification and doing example proofs. My results show how blocks in the blockchain function, and if its concepts are correct. We found that the use of public keys, signatures, addresses, transactions all function as expected.

Manual proofs in programming language are essential to have full coverage when it comes to proving concepts. However, you could also look into other automated methods for checking for correctness. SMT solvers deal with the problem of finding functional parameters that satisfy an equation [24]. This is just one solution to conducting manual proofs which are tedious, human error prone and time consuming. I do however believe Agda to be a very advanced functional language and Bitcoin would benefit greatly from a full modelling of Bitcoin. I believe that the question many cryptocurrency developers ask is if manual verification is needed. This study should inspire developers to incorporate some sort of verification when developing cryptocurrencies and the importance of verification.

6.1 The need for verification

Without verification we won't be able to test against attacks effectively, we went over majority of the attacks Bitcoin could be affected by. We also have the rising threat of quantum computing which in the future could play a huge role in exploiting cryptocurrencies.

6.2 Future Work

Number of proofs for modelling of Bitcoin is still incomplete. In my thesis I have focused on widening the specification of Bitcoin and also conduct a few proofs. My focus was the Bitcoin blockchain. There is however plenty more concepts in Bitcoin. In order to complete full modelling on Bitcoin, we need to first have a full specification of Bitcoin after completing this we can then start doing proofs on the specification. Concepts I believe should be modelled next would be mining, more examples of proofs on the specification would also be ideal.

Bibliography

[1] “What is Crypto Fear and Greed Index? Should you look at it before buying crypto?” (2022, January 18). [Online]. Available: https://tinyurl.com/2p8hwkzb

[2] A. Setzer. (2018, April 17). “Modelling Bitcoin in Agda”. [Online]. Available: http://arxiv.org/abs/1804.06398, arXiv:1804.06398

[3] “From Ecash to Ethereum—Where crypto was and where it’s heading today” (2022, March. 07). [Online]. Available: https://www.hindustantimes.com/brand-stories/from-ecash-to-ethereum-where-crypto-was-and-where-it-s-heading-today-101646653623078.html

[4] M. Raikwar, D. Gligoroski and K. Kralevska. (2019, June 20). "SoK of Used Cryptography in Blockchain,". [Online]. Available: https://ieeexplore.ieee.org/document/8865045

[5] A. Hayes. (2022, March 05). *Blockchain explained* [Online]. Available: https://www.investopedia.com/terms/b/blockchain.asp

[6] M. Nofer, P. Gomber, O. Hinz, et al. (2017, March 20). “Blockchain”. *Bus Inf Syst,* [Online]. Available: https://doi.org/10.1007/s12599-017-0467-3

[7] Simply Explained. (2018, March 21). *Proof-of-Stake (vs proof-of-work)* [Online]. Available: https://www.youtube.com/watch?v=M3EFi_POhps&ab_channel=SimplyExplained

[8] S. Nakamoto. (2008, October 28). “Bitcoin: A Peer-to-Peer Electronic Cash System”. *Decentralized Business Review* [Online]. Available: https://www.ussc.gov/sites/default/files/pdf/training/annual-national-training-seminar/2018/Emerging_Tech_Bitcoin_Crypto.pdf

[9] R Adams. (2022, April 15). *75 Cryptocurrency Statistics Show Crypto’s Gone Mainstream* [Online]. Available: https://youngandtheinvested.com/cryptocurrency-statistics/

[10] S. P. Gupta, K. Gupta and B. R. Chandavarkar. (2021, May 21). "The Role of Cryptography in Cryptocurrency". *2nd International Conference on Secure Cyber Computing and Communications* [online]. Available: https://ieeexplore.ieee.org/document/9478099

[11] “Agda (programming language)”. (2022, February 24). [Online]. Available: https://en.wikipedia.org/wiki/Agda_(programming_language)

[12] GitHub. (2022, Jan 20). *Agda* [Online]. Available: https://github.com/agda/agda/

[13] “Agda vs. Coq vs. Idris”. (2020, Feb 18). [Online]. Available: https://whatisrt.github.io/dependent-types/2020/02/18/agda-vs-coq-vs-idris.html

[14] “Coq”. (2022, March 18). [Online]. Available: https://en.wikipedia.org/wiki/Coq

[15] M. Apostolaki, G. Marti, J. Müller, et al. (2018, August 19). “SABRE: Protecting Bitcoin against Routing Attacks”. [Online]. Available: https://arxiv.org/abs/1808.06254

[16] Jake Frankenfield. (2022, April 15). *51% Attack* [Online]. Available: https://www.investopedia.com/terms/1/51-attack.asp

[17] K. Chaudhary, A. Fehnker, et al. (2015, November 13). “Modelling and Verification of the Bitcoin Protocol”. [Online]. Available: https://arxiv.org/pdf/1511.04173.pdf

[18] Y. Shahsavari, K. Zhang and C. Talhi. (2020, May 22). "A Theoretical Model for Block Propagation Analysis in Bitcoin Network". [Online]. Available: https://ieeexplore.ieee.org/document/9099002

[19] A. M. Antonopoulos, Mastering Bitcoin, 2nd ed. O'Reilly Media, 2017.

[20] “What is the block maturation time?”. (2012, November 14). [Online]. Available: https://bitcoin.stackexchange.com/posts/1991/revisions

[21] P. Ragde. *Logic and Computation Intertwined* [Online]. Available: https://cs.uwaterloo.ca/~plragde/flaneries/LACI/index.html

[22] “Language Reference”. [Online]. Available: https://agda.readthedocs.io/en/v2.6.0.1/language/index.html

[23] “Formal Specification and Verification”. (2020, July 17). [Online]. Available: https://why.cardano.org/en/science-and-engineering/formal-specification-and-verification/