Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Quartz for Cross-Platform Smart Contracts

Author:
Ali Chaudhry

Supervisor:
Prof. Susan Eisenbach

Second Marker:
Dr. Sergio Maffeis

June 15, 2020

Acknowledgements

بسم الله الرّحمن الرّحيم

الحمد لله رب العلمين والصلاة والسلام على رسوله محمد وعلى آله وأصحابه أجمعين

I would like to begin by thanking God for granting me the ability to complete this project especially in this time of global pandemic.

I would also like to thank:

- Professor Susan Eisenbach and Dr Sergio Maffeis for their guidance and support throughout this project
- My family for their supporting and guidance throughout my life
- My dear friends especially Zubair Mr Ghusl, Salih Baeyarni, Husaini and Abdulrahman for their help throughout the course of the project
- Lastly to my dear acquaintances in the Coup d'Isoc private members club for their riveting discussions regarding current affairs

Contents

Listings

Chapter 1

Introduction

The field of Blockchains has developed a lot of buzz in recent years, therefore has been the subject of both academic and commercial research and development. Many new approaches and blockchain technologies have been developed and taken shape in the last few years especially with the intention of facilitating decentralised financial applications. An important area within the blockchain scene is smart contracts, contracts that are written and deployed to live permanently on the blockchain that can be interacted with by other users or other smart contracts on the blockchain.

Flint [1] is a language that was developed in 2018 by Franklin Schrans with the intention of creating a language which allowed users to easily write smart contracts which are secure for the Ethereum [2] blockchain network. The need for such a language grew due to many cases of smart contracts written for the Ethereum blockchain being vulnerable to a number of attacks due to the complex semantics of Ethereum's smart contract programming language and hence being taken advantage of by malicious individuals. The security properties and design of Flint ensure that contracts that are safe from common vulnerabilities can be compiled to Solidity to then be deployed on the Ethereum blockchain.

Since the advent of Flint, others have also seen the need to be focus on security to enable the easier writing of secure smart contracts which can be deployed onto the popular blockchain platforms to prevent smart contracts from being exploited.

Facebook has also seen the potential of blockchain, developing their own blockchain technology platform Libra [3]. Designed to be a permissioned blockchain that supports digital currency for global transactions. Currently still in development and pending release Libra also supports smart contracts and Facebook are currently developing their own intermediate language for writing smart contracts Move IR.

After the announcement and partial release of the Move [4] intermediate representation language for Libra, it has become apparent that the current Flint language is very heavily focused on targeting the Ethereum blockchain and Solidity [5] language. Flint follows Solidity both in having an object orientated approach to the design of the language and the manner with which it handles currency. Move IR does not follow the same approach, instead following a more imperative design featuring modules and resources to handle currency which can be likened to abstract data types. It would currently be impossible to implement a compiler from the Flint language targeting Libra blockchain via compiling contract written in Flint to Move IR whilst not changing aspects of the language. This restriction is massive disadvantage to the Flint language that aimed on solving an issue that was not solely for a single target blockchain but rather to enable writing secure smart contracts for multiple targets however it cannot currently perform such a feat.

1.1 Contributions

We present in this project a language and compiler that is capable of targeting both the Ethereum and Libra blockchains by enabling compilation into the Solidity and Move IR languages respectively. This project provides the following contributions:

1. The Quartz Language:

    - This project presents the Quartz language, a language based off the original Flint language that allows the writing of smart contracts that can be leveraged to target both the Ethereum and Libra blockchain.
    - Interacting with External Contracts: The language introduces the ability to interact with external contracts, a major new feature that the original Flint language was incapable of doing limiting its usability. The ability to interact with external contracts enables users to write multiple contracts that after deployment can communicate with each other or other contracts on the blockchain to enable complex smart contract infrastructures.
    - Handling Cross-Platform Currency and Assets: The language introduces a new Asset construct that is inspired partly from the resource types from the Move IR language that were originally inspired by linear types. The new Asset construct allows for a more flexible approach of handling native currency on multiple blockchains. It also enables a more generic and flexible approach for creating and handling custom asset types.
    - Struct Passing by Value: The language introduces the ability to pass structs by value which is a fundamental feature that is used as a building block for the other work that has taken place to introduce new features to this language.

2. The Quartz Compiler:

    - This project presents the Quartz compiler, a working compiler written in the Rust[6] language that is capable of compiling smart contracts written in the Quartz language into smart contracts in the Solditity and Move IR languages. The compiler implementation is a rigorous piece of engineering work that follows an extensible architecture to allow for easier future extensions and additions.

1.2 Thesis Structure

The rest of the thesis is organised as follows: Chapter 2 provides a background overview of this report touching on the topics of blockchains and smart contracts. Chapter 3 delivers an introduction to programming in the Quartz language. Chapter 4 describes how Quartz enables interacting with external contracts. Chapter 5 describes the design of the new Asset construct after comparing the viable avenues of handling currency. Chapter 6 gives an in-depth explanation of the design and implementation of the Quartz compiler. Chapter 7 highlights the future extension work to be carried out on the Quartz language and compiler. Chapter 8 evaluates the outcome of the project, comparing the language and compiler to that of the Flint language and compiler. In Chapter 9, we conclude this report and project and touch upon the challenges faced during this project.

Chapter 2

Background

The aim of this section to introduce and explain to a sufficient level of detail the concepts required to understand what is involved in designing a language for smart contract programming. Furthermore, to expound upon some of the current approaches taken within this field highlighting specifically the method of encoding money or physical assets, evaluating these approaches to gain insight from their contributions.

2.1 What is a blockchain?

To provide a simplified generic notion of what a blockchain is, we will start by using the term used by most to explain the term blockchain. A blockchain is a type of distributed ledger. What is meant by this is it is a ledger, a record of transactions that have taken place between entities. Distributed, meaning that this record of transactions is held by many entities leading to the notion of a distributed ledger.

Taking the concept of what a distributed ledger is we can now begin to explain the specific features of a blockchain that distinguish it from other types of distributed ledgers. Fundamentally a blockchain is a sequence of blocks, each block containing a list of transactions as well as necessary header information such as the hash of the previous block. The collection of blocks is an add only data structure, it starts with the first block called the genesis block and as transactions take place new blocks are added on to this collection to form a chronological sequence of blocks.

This technology has been shown to have potential in providing a method of efficient record of activity that is almost tamper-proof. A lot of the focus of application has been in the financial industry with the advent of famous cryptocurrencies such as Bitcoin[7], Ether[2] and others. Cryptocurrencies leverage the blockchain technology concept with Bitcoin and Ethereum network both being blockchain implementations.

2.2 What is a smart contract?

The first coinage of the term smart contract was by Nick Szabo, who defines them to be "computerized transaction protocols that execute terms of a contract"[8]. Since the rise of blockchain technology and the different blockchain implementations, smart contracts now are more appropriately described as being a computer program that lives on the blockchain. A computer program that exists on the blockchain itself that other computer programs (smart contracts) or users can interact with.

Smart contracts are incredibly useful in the world of blockchain because they allow automatic

exchange of goods or currency between individuals as long as the terms for the exchange are satisfied. A computer program in the sense that if the specified terms within the contract are met by the parties involved, the execution of code can then occur which may be to make a transaction to someone. A key element to the surge of interest in smart contracts is the potential to automate many day to day interactions that usually would require a 3rd party or some centralised authority. An example of this could be an individual claiming a tax rebate, as long as the individual fulfils the conditions specified, in this case, maybe to provide sufficient identification information so the contract code can execute to calculate the tax rebate amount and process a transaction if a tax rebate is due.

As mentioned because Smart contracts are deployed on to the blockchain itself and the blockchain being an add only data structure once a smart contract is deployed it cannot be modified. Another consequence of their existence on the blockchain is that everyone on the platform can naturally see the contents of the smart contract due to transparent nature of every node being able to view every block of the blockchain. These properties of smart contracts are some of the key reasons why smart contracts are seen with such potential. The transparent, tamper-proof, efficient nature of them hold the potential to change the traditional complex processes today into efficient processes tomorrow.

It is this great potential for use in many real-life applications that has led to the development of platforms such as Ethereum. The Ethereum Project describes itself in the following manner "Ethereum is a decentralized platform that runs smart contracts". It is clear that is an area which not only has a lot of potential but is also experiencing a heavy focus of research and development.

After the release of the Ethereum network and rise of smart contract usage, it became clear that there was not enough focus on establishing the security of smart contracts before they were deployed. Several attacks took place against smart contracts on the Ethereum network due to mainly poorly written smart contracts.

These attacks and the fact that smart contracts cannot be modified after deployment has led to a lot of focus being emphasised on the establishment of correct behaviour of smart contracts especially. Many approaches have been taken concerning this goal, from designing new languages for programming smart contracts to developing newly designed blockchain platforms to achieve this. This now leads us onto the second part of this section where will be touching upon some of the approaches people have taken concerning language design for programming smart contracts.

2.3 Attacks on Smart Contracts

Since the inception of smart contracts, several well-known attacks have taken place on smart contracts especially on the Ethereum Blockchain. An extremely famous example is The DAO attack[9], on June 17th 2016 less than a month after the DAO contract was published it was attacked by an unknown individual. The attacker was able to extract 3.6 million Ether ($50 million) into his individual account. The attack was such a shock to the Ethereum community that it triggered a hard fork of the blockchain to return the stolen amount to the victims. What enabled the attack, was due to an oversight in the code of the contract allowing unintentional behaviour to be allowed.

Another example is that of The Parity Wallet Hack[10] in 2017 that led to $30 million to be stolen, the second-largest attack at the time. The attack was made possible because the initialisation function of the contract had no restrictions upon who can call the function.

As a result of the initialisation function not having any restriction on its caller and the payable code function delegating any non-matching calls to the library allowed the attacker to simply make himself the owner of the contract and then was able to call various other functions of the contract to transfer all the funds to himself.

The Ethereum Blockchain has a current market capitalisation of $23,767,314,199[11] with over 1.9

million contracts currently deployed on the blockchain, preventing attacks and ensuring security is a very important endeavour.

```
// constructor - just pass on the owner array to the multiowned and  // the limit
    to daylimit
function initWallet(address[] _owners, uint _required, uint _daylimit) {
  initDaylimit(_daylimit);
  initMultiowned(_owners, _required);
}

function() payable {
  // just being sent some cash?
  if (msg.value > 0)
    Deposit(msg.sender, msg.value);
  else if (msg.data.length > 0)
    _walletLibrary.delegatecall(msg.data);
}
```

The Ethereum Blockchain has a current market capitalisation of $23,767,314,199[11] with over 1.9 million contracts currently deployed on the blockchain, preventing attacks and ensuring security is a very important endeavour.

2.4 Smart Contract Security approaches

There seem to be 2 paths taken by the blockchain and developer community concerning establishing the security of smart contracts. The first path being in the area of verification of smart contracts. Tools have been developed to establish the correctness and safety of contracts written in Solidity and other native blockchain languages. The second path has been in the area of smart contract language design. After, realising that a number of the attack vectors were as a result of language design choices within Solidity and its aim of being Turing complete and efficient. The community has made strides towards the development of alternative languages with a stronger focus on security to prevent programming oversight from happening in the first place.

2.4.1 Smart Contract Verification

Initially, verification tools focused on scanning Ethereum contracts for well-known vulnerabilities and bad programming practices, the focus was on static analysis of the smart contracts. Tools like Securify[12] and SmartCheck[13], fall into this category providing pre-deployment auditing of smart contracts. Recently, strides have been made towards full formal verification of smart contracts to provide the best possible guarantees to smart contract programmers.

Verx

VerX[14] is an automatic contract verifier for custom contract requirements. Created by the same organisation as the Securify tool, Chainsecurity[15] the tool provides complete formal guarantees regarding a smart contract. VerX works in the following manner, the user provides the tool with a smart contract and a technical specification to match written in VerX's specification language. The tool then verifies whether the requirements in the specifications will or will not hold in a given contract.

Complete verification of smart contracts especially on the Ethereum blockchain is a very recent development, previously the only option was post-production tools either analysing to check for known vulnerabilities or generic code analysis techniques such as Fuzzing and Dynamic Symbolic Execution. The great advantage of this approach, especially with the onset of tools such as VerX, is a smart contract programmer can be assured of complete specification correctness of their contract. The other path of language design as we shall see does not necessarily have these complete

guarantees about contract correctness. However, a downside to this approach is a smart contract programmer has to know and be able to correctly produce specifications in the format the tool requires.

2.4.2 Smart Contract Language design approaches

The second approach and the one that is going to be analysed more heavily is that of language design for smart contracts. Designing languages that address the concern of having clear semantics and that provide minimum guarantees concerning security.

Flint

The first approach to analyse is that of a language called Flint, Flint was designed to be language to enable the programming of smart contracts for the Ethereum platform. The main motivating factor behind the Flint language was due to the need to establish behaviour correctness before deployment of a contract on to Ethereum because it cant be modified post-deployment. The Ethereum platform already had a language for programming smart contracts, Solidity an object-orientated language. However, Solidity as a language did not have a clear focus on security or behaviour correctness for smart contracts, but more of a focus on the expressiveness of the language. This led to many vulnerabilities and methods of attack against Solidity contracts throughout its lifetime and has caused several necessary revisions to take place upon the original language. Flint aims were to prevent attacks by incorporating features to establish smart contract security.

One of the reasons current smart contracts written in Solidity were subject to vulnerability was due to the non-trivial nature of the syntax and semantics of the language. To tackle this problem Flint aimed to design the syntax in a manner to allow "programmers to write and reason about smart contracts easily." The Flint implementation was in Swift and it also was based on the Swift language concerning syntax.

With regards to the semantics of the Flint language to fulfil the purpose of easily reasoning about the smart contract, in Flint, there is a clear separation between what is considered the state of the contract and that which can cause changes to that state, the functions of the contract. This has some clear benefits for the user of the language, it becomes much easier to reason about what state is needed for the contract as well as what possible mutations to that state can occur and how exactly those mutations can take place. Eliminating the possibility of pieces of state which are not deemed necessary by the programmer from the contract.

Another important feature the Flint language implemented is its "Calling Conventions". One of the desired qualities of a smart contract is the ability to have access control over the functions to prevent any unintended mutations to the contract. Flint achieves this by making the functions of a contract be declared within a "Behaviour declaration block." The smart contract programmer must decide and declare the access control of the functions within that contract.

After providing a small explanation upon some of the features of the Flint language, it becomes clear that the mantra driving a lot of the design decisions of the language was the idea of making it as easy as possible for a programmer to write a smart contract whilst having features that nudged the programmer into considering specific elements of their contract that would have consequences for security which were being overlooked by smart contract programmers in solidity. Below is an example contract that is implemented in Flint to highlight this key separation of state and mutation of state taken from the Flint website.

```
// This is the declaration of the contract. In this simple example, it only
// includes the one state variable, `hits`.
contract Counter {
  // `hits` will initially be `0` and it will be an integer variable (`Int`).
  var hits: Int = 0
}

```

```
// These are the functions of the contract. The `:: (any)` indicates that
// these functions can be called by anyone on the Ethereum network.
Counter :: (any) {
  // This is the constructor, called when the contract is first created. There
  // is nothing we need to do here at this point, so it is empty.
  public init() {}

  // This function returns the current counter value. It takes no arguments,
  // but returns an `Int`.
  public func getValue() -> Int {
    return hits
  }

  // This function increates the counter value by one. It only does this if
  // some Wei was paid, so the function is `@payable`. The amount of Wei paid
  // is available as the implicit `value` argument, although we do not use this
  // value here.
  @payable
  public func hit(implicit value: Wei) mutates (hits) {
    hits += 1
  }
}
```

Listing 2.1: Counter Flint Contract Example

The contract has 2 declaration blocks, one as shown at the top of the contract only containing members of state for the contract, in this case, the number of hits. Followed by the function declaration block to implement that which can modify the state of the contract. Another thing to note in this example is the use of (any) at the beginning of function declaration block to set a contract level accessibility level, this is precisely the Calling Conventions[1] feature mentioned earlier.

Encoding Money

The approach Flint took towards the task of encoding money or currency is by creating an Asset trait that users can implement on their struct constructs for their custom assets. A trait is similar to an interface in the traditional object orientated programming sense, essentially a set of functions that a struct is required to implement if that particular struct is of that trait. In the case of a custom asset, such as a ticket a user would define a struct Ticket which implemented the Asset trait ensuring the ticket struct has an implementation of the set of functions defined in the trait Asset. Flint provided users with a building block for their own custom currencies or assets. The standard library for Flint has an implementation for Wei, the smallest denomination of Ether.

Below is the Flint implementation for the Asset trait that is used to allow the implementation of currency. The Wei currency implemented in the standard library of Flint also uses the Asset trait.

```
struct trait Asset {

  // Initialises the asset "unsafely", i.e. from `amount` given as an integer.
  init(unsafeRawValue: Int)

  // Initialises the asset by transferring `amount` from an existing asset.
  // Should check if `source` has sufficient funds, and cause a fatal error
  // if not.
  init(source: inout Self, amount: Int)

  // Initialises the asset by transferring all funds from `source`.
  // `source` should be left empty.
  init(source: inout Self)

  // Moves `amount` from `source` into `this` asset.
  func transfer(source: inout Self, amount: Int)
  //    post (source.rawValue == prev(source.rawValue) - amount
  //            && self.rawValue == prev(self.rawValue) + amount
  //            || self == source
  //    )
  {
```

```
22      if source.getRawValue() < amount {
23        fatalError()
24      }
25
26      let unused1: Int = source.setRawValue(value: source.getRawValue() - amount)
27      let unused2: Int = setRawValue(value: getRawValue() + amount)
28    }
29
30    func transfer(source: inout Self)
31    //  post (source.rawValue == 0 || source == self)
32    {
33      transfer(source: &source, amount: source.getRawValue())
34    }
35
36    // Returns the funds contained in this asset, as an integer.
37    func setRawValue(value: Int) -> Int
38
39    // Returns the funds contained in this asset, as an integer.
40    func getRawValue() -> Int
41 }
42
```

Listing 2.2: Asset Trait Implementation

Libra

The next approach to evaluate is that taken of Facebook[16] in its development of the Libra blockchain and its corresponding Move IR language. Facebooks aim behind the Libra blockchain is to enable easy global access to financial services especially targeting those regions where the access is not currently possible. The development of the Libra blockchain and the Libra currency is key to achieving these goals. Facebook chose to create a new programming language for the Libra blockchain to handle smart contracts and transactions called Move.

One of the key advantages Facebook had when designing their own blockchain implementation Libra as well as the official language Move was the experience of everyone in the industry before them. Adding on to this they were designing their own blockchain implementation therefore had the flexibility of being in control of all the design elements and features. These advantages become clear when we begin to look at some of the features of Libra that were incorporated as part of their system.

A difference to note with respect to the Libra network is that contrary to the Ethereum network it is a permissioned blockchain. The former being permissionless, therefore anyone can join the blockchain whenever they want. Being a permissioned blockchain, to join the blockchain requires the permission of certain authoritative entities.

Facebook were focused on from the onset on targeting the problem of digital encoding of assets in a manner which previous blockchains did not specifically focus on. That is the encoding of physical assets in a digital manner that has a heavy emphasis on preserving the scarcity and access control properties of those assets. This being tied into Facebook's vision for Libra being a hub for global exchange with a global currency Libra.

Libras programming language which enables smart contract like functionality is known as Move IR. Move Intermediate Representation which differed from previous languages not leaning towards an object orientated design approach but more so towards an imperative approach. Moves equivalent of a smart contract is a Module and currency as well as assets are implemented using Resource types. It's both the Move Module and Move Resources which allow for the codification of a smart contracts state and operations which alter that state.

Modules

Being the equivalent of smart contracts, the modules allow the programmer to set the rules for the manipulation of the types defined within the module. Very similar to the approach of Flint Move modules have a separation between the declared state of the contract and the operations that can be performed on said contract.

Here we present a snippet of the implementation of the LibraCoin module from the Libra source code to make this code construct. LibraCoin is the official proposed currency for the Libra network. First focusing on the declarations of state for the contract we can see 3 struct declarations namely, for the LibraCoin, the mint capability of the LibraCoin and the MarketCap for the LibraCoin. It is these resource structs that are used in Move to declare state for the module and allow for the custom resource types that enable digital encoding of physical assets in Move.

After the struct declarations inside the module, all the procedures (operations) that can take place as part of the module are declared. In this example we have omitted procedures linked to minting to allow for a simpler example. The 3 main focuses when handling currency or assets is to prevent unauthorised creation, duplication or destruction. Move resources are not able to be reused, duplicated or discarded. They can only be created or destroyed by their defining module. Destruction of a non-zero LibraCoin is prevented by virtue of the destroy_zero function, being the only way to destroy a coin which fails if non-zero.

```
module LibraCoin {
    use 0x0::Transaction;

    // A resource representing the Libra coin
    // The value of the coin. May be zero
    resource struct T { value: u64 }

    // A singleton resource that grants access to `LibraCoin::mint`.
    // Only the Association has one.
    resource struct MintCapability {}

    // The sum of the values of all LibraCoin::T resources in the system
    resource struct MarketCap { total_value: u64 }

    // This can only be invoked by the Association address, and only a single time.
    // Currently, it is invoked in the genesis transaction
    public initialize() {
        // Only callable by the Association address
        Transaction::assert(Transaction::sender() == 0xA550C18, 1);

        move_to_sender(MintCapability{});
        move_to_sender(MarketCap { total_value: 0 });
    }

    // Return the total value of all Libra in the system
    public market_cap(): u64 acquires MarketCap {
        borrow_global<MarketCap>(0xA550C18).total_value
    }

    // Create a new LibraCoin::T with a value of 0
    public zero(): T {
        T { value: 0 }
    }

    // Public accessor for the value of a coin
    public value(coin: &T): u64 {
        coin.value
    }

    // Splits the given coin into two and returns them both
    // It leverages `Self::withdraw` for any verifications of the values
    public split(coin: T, amount: u64): (T, T) {
        let other = withdraw(&mut coin, amount);
        (coin, other)
    }

    // "Divides" the given coin into two, where original coin is modified in place
```

```
    // The original coin will have value = original value - `amount`
    // The new coin will have a value = `amount`
    // Fails if the coins value is less than `amount`
    public withdraw(coin: &mut T, amount: u64): T {
        // Check that `amount` is less than the coin's value
        Transaction::assert(coin.value >= amount, 10);
        // Split the coin
        coin.value = coin.value - amount;
        T { value: amount }
    }

    // Merges two coins and returns a new coin whose value is
    // equal to the sum of the two inputs
    public join(coin1: T, coin2: T): T  {
        deposit(&mut coin1, coin2);
        coin1
    }

    // "Merges" the two coins
    // The coin passed in by reference will have a value equal
    // to the sum of the two coins
    // The `check` coin is consumed in the process
    public deposit(coin: &mut T, check: T) {
        let T { value } = check;
        coin.value = coin.value + value
    }

    // Destroy a coin
    // Fails if the value is non-zero
    // The amount of LibraCoin::T in the system is a tightly controlled property,
    // so you cannot "burn" any non-zero amount of LibraCoin::T
    public destroy_zero(coin: Self::T) {
        let T { value } = coin;
        Transaction::assert(value == 0, 11)
    }

}
```

Listing 2.3: LibraCoin Module Implementation example

From a programmers perspective by default they have protections in place as part of the language and system implementation that their defined currency or assets are not subject to certain vulnerabilities. The objective of the smart contract (module) writer is declaring state of the contract, how that state can change and creation or destruction of their resources.

The implementation for the enforcement of the properties is done via the bytecode verifier[4], which prevents any publication or execution of Move programs without first being verified. This holds the advantage of course that all smart contracts on the Libra network will subject to such checks and will benefit from such properties.

This approach seems to be intuitive for the most part, providing system security features as well as giving some control to the programmer to define rules for aspects such as creation and destruction. It differs from the approach of Flint relies on the user using the Asset trait specifically to get the security properties required where as these apply for all resources whether they be a monetary one or not.

2.4.3 Scilla

A fascinating approach to the development smart contract programming languages is that of Scilla[17]. An intermediate functional style language with a heavy focus on formal verification of contract behaviour. The Scilla language was designed and developed for the Zilliqa blockchain.

Built upon System F leveraging the tools existing in the field for formal verification, Scilla uses the Coq[18] proof assistant in order to perform its formal verification. The design took an approach of minimalism building from the ground up to achieve a language that was built with formal

$$\tau \ ::= \ \textbf{Money} \mid \textbf{NotMoney} \mid \textbf{Map}\ \tau \mid t\ \overline{\tau} \mid \top \mid \bot$$
$$t \ ::= \ \textbf{Option} \mid \textbf{Pair} \mid \textbf{List} \mid \ \ldots$$

(maps)	$\textbf{Map}\ \tau \sqsubseteq \textbf{Map}\ \tau'$	iff $\tau \sqsubseteq \tau'$
(algebraic types)	$t\ \overline{\tau} \sqsubseteq t'\ \overline{\tau'}$	iff $t = t'$ and $\tau_i \sqsubseteq \tau'_i$ for all i
(bottom)	$\bot \sqsubseteq \tau$	for all τ
(top)	$\tau \sqsubseteq \top$	for all τ

Figure 2.1: Defintion of Tag type

verification in mind from the roots. A clear stance that was adopted was the high expressiveness of languages currently designed for smart contracts such as Solidity are one of the clear factors that led to the deployment of vulnerable contracts. Scilla specifically reduced the level of expressivity within the language to increase the safety of the language. By having greater minimalism it is easier to reason about and formalise the semantics of the entire language.

Two major fundamental decision principles sit at the bedrock behind the design of the Scilla language to facilitate formal verification of contracts written in Scilla. The first being a clear separation between any computation that takes place within the contract and any communication the contract partakes in with other entities. The second is another clear separation between computations that mutate the state of the contract and those that do not.

Scilla breaks down smart contracts to 4 fundamental components developing a verified specification for each component. The 4 core components of a smart contract are Communication, State Manipulation, Effects and Computations.

These decisions are what lead to Scilla being a statically typed functional language which models the smart contract as a state machine and any computation upon the state machine as a transition. This method of modelling smart contracts and computations that it partakes in leads to an intuitive method of formally specifying the semantics for the Scilla language. This approach is tremendously useful correct behaviour verification of contracts before deployment.

Understanding a smart contract within the Scilla ecosystem would be to consider a contract to be a state machine which is infinite, any computation is a transition of state leading the contract to a new state. The communication element refers to any interaction between contracts. A contract consists of first 2 sets of state, the first being all the mutable parts of the contracts state and the immutable parts. Secondly, 2 sets defined computation on the contract, that which are all pure functions and those mutable functions that change a contracts state. This methodology mirrors closely those of the Flint and Libra approaches which seem to make clear that an integral part of designing a smart contract language is providing a structure where the smart contract programmer is put in a position where each of these elements are considered separately.

Encoding Digital Money

A major contribution of the work is the approach taken to handling native and non native currency or tokens. Scilla implemented a cash flow analysis as part of their checking framework, able to determine which elements of a contracts state that are 'Money' and 'NotMoney'.

We will now give a simplified explanation of how the cash flow analysis is performed to evaluate the approach taken to handling digital encoding of money.

First we explain the existence of Tags τ essential to the cash flow analysis. The tags in this analysis is simply an abstract data type that is defined as the following, as well a partial ordering over on the tags.

As can be seen from the definition tags either point towards Money, NotMoney, Nothing, a datatype

or map over datatype and an Inconsistency tag state.

The cash flow analysis begins by tagging every element of the contracts state with the a tag ($\perp$). A key exception being 2 implicit fields of the contract state, balance and accountAddress which are tagged as Money and NotMoney appropriately.

Having defined the formal semantics for the operations between elements of a contracts state, parameters or variable or the like an top down analysis of all the expressions within a contract can be done. The top down analysis of the usage of variables and other elements of state can be done to an expected tag value for that element.

An easy example to make clear how this would work would be to take the addition operation, as part of the formal semantics the 2 parameters to the addition operation are expected to be of same tag. Therefore a Money tagged parameter can be successfully added with another Money tagged parameter but not with NotMoney tagged parameter.

Having default parameters within the contract which are initially labelled as Money and NotMoney, allows the analysis to analyse all the interactions of environment state with other pieces of state. Anything interacting with those initial fields of Money in a manner only Money can interact with them are labelled Money and likewise with NotMoney. If after the complete analysis there is consistent labelling of tags then all the interactions are considered consistent with the semantics and it should be the case that those pieces of state which represent some Money object be appropriately tagged. If it is the case that for example the add operation has 2 parameters, one being Money and the other NotMoney, then the result of the interaction is inconsistent and not compliant with the formal semantics.

One of the limitations of this approach as mentioned in the Scilla paper is that when handling Currency/Assets/Tokens which are non native that which differ from the official currency of the blockchain network, if they do not interact with the initial fields tagged with Money NotMoney it is not possible to be able to actually recognise them as money objects.

It would require the user to encode some sort of interaction with those initial fields so that the cashflow analysis can actually run properly over the non native assets.

An approach which differs greatly from the others mentioned where the focus has been creating some sort of core part of the language which is to be used to encode digital money or assets instead allowing the programmer the freedom to define non-native tokens as they wish. The downside being if the programmer wants the cash flow analysis to successfully tag the token as Money it is required to somehow ensure its interaction with another that will be tagged by Money.

2.4.4 Remarks

All the approaches mentioned share the principal concept in their language design to create a separation in the language itself between the contract properties and that which causes the contract to change, it's functions. It seems clear that is necessary to force the programmer to have this separation in mind whilst creating the smart contract preventing any unintended effects from taking place after deployment.

An interesting difference however is how each approach takes a slightly different methodology for language style. Flint going for a more object orientated approach, Libra an imperative one and Scilla an almost pure functional one. These choices seem to be heavily influenced due to the features each language desires to provide to the user.

Reflecting on the features each language provides and the design choices they chose we will summarise the contributions of each language and the what those contributions means for the field of smart contract programming languages.

The Move language has 3 characteristic features that distinguish it's approach, limited expressivity

to aid more secure coding, features such as dynamic dispatch are not supported. Secondly, the introduction of a new resource type which allows the platform to give the guarantee that local invariants become global invariants when handling a resource. Finally, the Move compiler performs type checking on the bytecode level to ensure well typed bytecode as opposed to Solidity's approach of a well typed program that is then compiled to the bytecode, this provides safer guarantee regarding type correctness of final bytecode.

The Scilla language, provides a non-turing complete language that heavily focused on being able to be formally verified as part of the compilation process with their embedded proof checker. It enforces a separation of communication and computation to allow for better verification of functionality.

Although, the other languages may offer better security by design both have not been tested with real deployed contracts to the extent that the Solidity language has.

The Move language has not been officially released yet, therefore it will be necessary to observe after the release to confirm if the proposed methodology of security by reduction of expressivity holds up well in the public domain. The guarantee that the Move language provides of a local invariant becoming a global invariant does not provide any guarantee of functional correctness for resource conservation because user programming mistakes are not caught hence allowing local invariants to be affected by those mistakes. The Scilla language's choice to not have Turing completeness may limit the possibilities and usability of the language which needs to be observed as the language's use rises after the launch of the Ziliqa blockchain.

The Solidity language benefits from being the first majorly used smart contract programming lnaguage and hence benefits from a range of tooling that perform not only static analysis but also formal verification of Solidity programs.

Analysing all three of the languages, in theory the Move and Scilla languages appear to be much more secure than Solidity. However, when taking into account the tooling available for each of the languages formally verified programs are either available or in-progress of being supported. Therefore, the overall difference between what each language and it's tooling provides do not differ much. However, it is clear that the blockchain community will lean to either the first approach of having expressivity and using tooling to provide security or the approach of limited expressivity to provide better security from the design stage. It is clear, that there is unfilled gap in the field for a language which provides ease of use with expressivity whilst still providing strong security guarantees.

Chapter 3

Quartz Language

3.1 Overview of Quartz

We present the Quartz language, a language designed to incorporate the original aims of the Flint language, providing security and ease for the programmer but also with the ability to write contracts targeted for multiple platforms. The Quartz language in conjunction with the implemented Quartz compiler is capable of producing contracts that target both the Ethereum and Libra blockchain platforms. The Quartz language also provides new features such as the ability to interact with external smart contracts, struct passing by value and it provides a notion and representation for handling currency using the Asset construct which works across different platforms.

For the rest of this chapter, we will give an introduction to the language. Firstly looking at examples of how contracts and their functionality are coded in the Quartz language, highlighting the revisions that have taken place from the Flint language both in syntax and semantics. The chapters following will proceed to explain in further detail the major new features that the Quartz language oers.

3.1.1 Encoding Contract State and Behaviour

The major aim of the project was to provide a language that enables smart contracts to be compiled for the Ethereum and Libra blockchain. Therefore, a huge part of the design of the Quartz language was investigating what parts of the original Flint language which already compiled down for the Ethereum platform, could be compiled into Move without any language redesign. A large portion of the language itself has not changed from Flint since it was possible to create a working compiler to the Move language for those parts of the language. There have been syntax and semantic changes to the already existing elements of the language, which will be explained further. A large portion of the contribution comes in the form of new features that are explained in the sections to come. The BNF for the Quartz language can be found in the Appendix with revisions highlighted, green indicating that which is completely new and blue indicating modifications made to existing constructs within the Flint language.

A clear key difference from the BNF's of the Quartz and original Flint language is the range of top-level declarations within the language, we will quickly explain the purpose each one serves and then proceed to give a small introduction into how contracts are programmed in Quartz. Originally, Flint only had a concept of a ContractDeclaration where the state would live, ContractBehaviour-Declaration where contract function logic would live and StructDeclarations, the declaration of structs properties and processing logic. Due to extensions provided to the Flint language since the advent of Flint and due to work done in this project the list of top-level declarations is as follows.

1. Contract Declaration: The Block containing all contract state
2. Contract Behaviour Declaration: A set of contract logic annotated with specified caller restrictions
3. Struct Declaration: A complex type data structure containing both state and logic for itself
4. Enum Declaration: A group of values sharing a common type
5. Trait Declaration: The specification of the external interface of an external contract or module
6. Asset Declaration: A complex type that is used to model currency and asset types

The Flint language acted as more as a basis than originally intended due to the fundamental parts of the contract writing grammar not needing to change to target the Libra language and instead could be handled within the compiler. In order, to keep the Quartz language usable and easy to use for new smart contract programmer, we also enforce a separation between state and computation similar to of the Flint, Move and Scilla languages. As can be seen in the example below, the state properties of the contract have to be declared within the contract declaration block and the functionality of the contract has to be declared within a contract behaviour declaration block. This enforcement of separation has some likeness to the stricter separation of logic and state enforced in Move modules and therefore was an intuitive approach to keep for the language, both from a security by design and practical implementation lens concerning being able to target multiple target languages.

We use the following example of the Shapes contract, to show how a basic contract is written within the Quartz language

```
1  contract Shapes {
2    var rectangle: Rectangle
3  }
4
5  Shapes :: caller <- (any) {
6    public init(rectangle: Int) {
7      self.rectangle = Rectangle(
8          width: 2 * rectangle,
9          height: rectangle
10     )
11   }
12
13   public func area() -> Int {
14     return rectangle.width * rectangle.height
15   }
16
17   public func semiPerimeter() -> Int {
18     return rectangle.width + rectangle.height
19   }
20
21   public func perimeter() -> Int {
22     return 2 * semiPerimeter()
23   }
24
25   public func smallerWidth(otherRectWidth: Int) -> Bool {
26     return self.rectangle.width < otherRectWidth
27   }
28 }
```

The Shapes contract declares a single state property of type Rectangle inside the contract declaration as can be seen in lines 1-3. The Rectangle type refers to a struct that will be shown in the upcoming example. From lines 5 - 29, the contract behaviour declaration contains a set of functions. The functions and initialiser of the contract are all within an any caller block, the behaviour declaration block is annotated with the caller protection any, therefore there is no restriction on the caller of any of the containing functions. The ability to place restrictions on the callers of a function that was enabled by the Caller Protections feature in the original Flint language has

been preserved in the Quartz language since it is a fundamental feature when targeting secure by design smart contract writing. The init(rectangle: Int) contract initialisation function takes a single Int argument and initialises the contract state by assigning to the rectangle state property an instance of the Rectangle struct. The area function of contract, returns back an Int value after calculating the multiple of both the width and height properties of the rectangle state property. The rectangle.width syntax allows the contract to access the property of a complex type. The functions following provide similar functionality providing data after performing processing on the internal rectangle state property.

The typestates feature and the contract traits feature which were present in the Flint language are not currently supported for the Quartz language. The typestates feature allowed the user to assign a list of possible states the contract can be in and subsequently assign allowed states to a contract behaviour block. Therefore, restricting what functions can be invoked depending on current typestate of contract. Currently, we do not have a replacement for the typestates feature however the same functionality can easily be replicated by the user using a state variable and an assertion statement to check if the contract is in the correct state before executing function. The contract trait feature granted the user to define an interface for the contract so that the contract matches the specified interface format. Currently, there is no replacement for contract traits in Quartz however the feature itself does not diminish the language's ability to write smart contracts compared to Flint.

3.1.2 Struct Encoding

The Quartz language allows the user to declare and manipulate structs as complex types. The following code shows the struct declaration for the Rectangle struct. At the top of the declaration we have the state properties for the struct defined, followed by the initialisation logic and then the additional functional logic.

```

struct Rectangle {
  public var width: Int
  public var height: Int

  public init(width: Int, height: Int) {
    self.width = width
    self.height = height
  }

  public func diagonal(wideness: Int = width, tallness: Int = height) -> Int {
    return (wideness ** 2 + tallness ** 2) ** 0
  }
}
```

The initialisation function has to specify how the state properties are to be instantiated when a new instance of a struct is created, as shown in the previous code above. Structs when stored as state properties on the Ethereum platform reside in the state for the contract, either in the EVM storage. However, Modules in Move do not have state in terms of an actual memory storage. Instead the state of the contract is stored within a resource that is stored as part of the storage of a Libra account. The state is stored in the accounts resources for the account that the Quartz contract is published on. Struct functions cannot be overloaded like contract functions.

The struct available in the Quartz language differs from the Flint language struct in 2 main ways. Firstly, the Quartz language supports struct passing by value on the Libra platform giving us a more powerful version of a struct than that of the Flint language. The second difference is that struct trait declarations as a feature were removed in the Quartz language. The feature allowed users to specify an interface for a struct that the struct had to match. The main use case of the feature was having a struct implement an asset trait as a means of guidance for encoding currency. This has been replaced with the new asset construct that we introduce in Chapter 5.

3.1.3 Enum Encoding

To increase the expressivity of the language we allow users to define custom enums. Enum declarations allow for the user to define a group of values of a common type. By allowing users to define their custom enums we provide the further ability for users to work with high-level language features in a manner that is safe and controlled. We illustrate a Quartz enum in the code example below.

```
enum State: Int {
  case locked
  case unlocked
}
```

The enum is declared with a corresponding type and each of the cases are listed below. It is possible to give each of the cases a raw value if a raw value is provided for a case the language will try to infer the values of the cases proceeding it based on the raw values preceding it. We illustrate this with an example below and demonstrate how an enum is used by the user.

```
enum State: Int {
  case locked = 0
  case unlocked
}
```

The second case will be inferred to have a raw value of 1 based upon the preceding case's value. The enum feature has not changed from the Flint language in the semantical or syntactical definition.

Accessing and using the enum cases is shown below where a variable can be declared with the type of the Enum identifier. The compiler handles converting this to the relevant type at the code generation stage if needed.

```
var state: State
state = State.locked
```

This feature was not present in the original Flint language and was introduced later as part of further extension work. The feature we present is the same as the one that was introduced in the extension work we have seen no reason to make any modifications to the feature and believe it is an extremely useful feature to have within the language.

3.1.4 Assets and Traits

Asset and trait declarations are two of the major new features introduced in the Quartz language. The asset construct is how the Quartz language handles cross-platform currency and allows encoding of assets. Trait declarations provide the ability to interact with external contracts. Both of these new features are illustrated and explained in greater depth in the next 2 chapters of this report.

3.2 Types

Quartz is a statically typed language, that supports both a set of primitive and complex types. Primitive types by default are passed by value. Complex types can be passed via references or by value. Traits on structs were a feature that was added later to the Flint language by future contributors but we do not currently support traits on Structs in the Quartz language. This is mainly because of the introduction of the Asset construct which has replaced the main previous use case of traits on Structs. Here is a table showing the internal types the Quartz language has.

Type Variant	Type	Description
Primitive Types	Int	256 bit unsigned integer
	Bool	Boolean value
	String	String value
	Address	160 bit Ethereum Address or 256 bit Libra Address
Complex Types	Array	A dynamically sized list of elements of type T. Elements can be added to it or removed from it.
	Dictionary	A dynamically sized mapping of key to value, where key is of type Address and v is of type V.
	Structs	Struct values, including user defined Structs are considered dynamic types.
	Assets	Asset values, including user defined Assets are considered dynamic types.

The internal types for the Quartz language have experienced some revision, mainly of a semantic nature. It is necessary to differentiate between the semantic definition of an Address internal type when the compiler is used for each target because of the platform differences of Ethereum and Libra. So the compiler will accept a hex address value of up to 256 bits to meet the requirements of the Libra platform but then the semantic analyser will perform the check to make sure any static values of Address meet the requirements of each platform.

The Quartz language has support for both arrays and dictionaries, supporting them for the Libra platform required work to provide such functionality due to the lack of high-level data structures in Move, the work to support them for the Libra blockchain is explained in Section 6.6.3. Dictionaries are restricted in that the key value has to be of type Address. This restriction is present due to specifics of the implementation used to support a dictionary data structure, explained in the section mentioned above. We feel that having dictionary support for Address to Type mappings is sufficient for a large if not most use cases of smart contracts.

The fixed-size array that was present in Flint is currently not supported but can be supported in the future, the implementation work required for such a feature is explained further in Section 6.6.3.

The Quartz language has introduced it's version of an Asset construct and hence being a variant of the Struct type it is a new complex type within the Quartz language.

3.3 Standard Library

We have implemented a standard library consisting of functions that can be used by the user in their contracts. The standard library for the Ethereum and Libra targets are separate and will be summarised below each in their respective table.

3.4 Concluding Remarks

We have given an introduction to programming in the Quartz language touching upon some of the fundamental building blocks for a contract within the Quartz language detailing what has changed from the Flint language and what has been used as a basis for this language. In conclusion, the

building blocks of the language such as the separation of state and computation have remained the same whilst the level of expressivity and language features have experienced changes. The chapters following illustrate and expound upon the new ability to interact with external contracts and the new Asset construct.

Chapter 4

Interacting with external contracts

We have introduced the ability to interact with external contracts within the Quartz language for both the Libra and Ethereum platform. In this chapter, we will explain the design, implementation and present an example use case for this language feature.

4.1 Motivation

Interacting with external contracts is one of the major selling points and features that makes smart contracts on the blockchain platform so useful and promising. However, the Flint language that Franklin presented did not have any capability of handling this limiting the usability of it and it's contracts. There was an attempt previously to carry out work on the Flint language to allow some form of external calls. As part of this project, we provide the ability to perform external calls for both the Ethereum and Libra platforms.

4.2 Design and Implementation

Interacting with external contracts is enabled through the use of two language features, traits which are essentially interfaces for the external contract and secondly external function calls enabling the calling of external functions. The trait feature takes inspiration from Rust Traits [19]. A trait is simply a block consisting of the function signatures that are present on the contract being specified, very similar to the idea of an interface of a class. Internally, as part of the standard library, a trait is used to allow interaction with the Libra platform's native currency. Flint previously had traits for structs which were an interface the struct had to satisfy, they were removed in the Quartz language, further explained in Section 5.4. Part of supporting interacting with external contracts involved documenting their public interface and for that, we introduce the trait feature which is inspired in its syntactic design from the legacy struct traits feature that was removed in the Quartz language.

Here we present two examples of a List trait representing an external List contract, the first being one for an external Ethereum smart contract and the second for an external Libra module. The list trait, has 3 function signatures specified. A key difference to note is the @contract and @module annotations above each of the traits, the explanation of this difference is given in the following paragraph. Focusing, on the example the list trait gives an interface to interact with 3 functions, 2 with non-void return types and 1 void function.

```
1 @contract
2 trait List {
3
```

```
  public func get_list_length() -> uint64
  public func is_empty() -> bool
  public func push(n: uint64)

}
```

Listing 4.1: Ethereum Trait example

```
@resource
@module(address: 0x33138303ce638c8fa469435250f5f1c3)
trait List {

  public func get_list_length() -> uint64
  public func is_empty() -> bool
  public func push(n: uint64)

}
```

Listing 4.2: Ethereum Libra example

When declaring a trait it is necessary to annotate the trait with the appropriate modifier due to platform differences between the blockchains. The modifier informs the compiler what type of trait is being declared and also how to handle the processing of the trait and its interactions with the smart contract. On the Ethereum platform, the only possible trait is one on an external contract, and the return types of the functions on that Ethereum contract are limited to the Solidity types which differs very much to the Libra platform. It is possible to have a trait on an external module, resource or struct on the Libra platform, the nuanced differences will be touched on later but it is also important to note that the functions on external modules can return resources and structs as well as the standard primitive types such as Solidity. The reason for the difference in the notation for the modifiers between the Ethereum and Libra examples is due to a massive difference in platform semantics. In the Ethereum blockchain, the code and state of a contract live on the same Ethereum platform address therefore we can specify the address when initialising the instance of the trait as shown in the example in 4.4. However, on the Libra blockchain, the code and the state of a module have their own address, which is not necessarily the same. In the case of traits in Quartz, we need to know the address of the module because it is necessary to facilitate importing the module's code into the Move contract. However, since the state of the contract has a separate address this necessitates knowing the intended address when initialising the trait instance because the Move contract has to know on what address to initialise an instance of the trait's state. The @resource modifier is needed to inform the compiler that the trait requires an instance of the module's state and that the function calls to the external contract have to be transformed by the compiler to include the address used for the instance as the first argument. These differences result due to the fact a smart contract on the Ethereum blockchain is a single instance of a contract deployed on the blockchain with its own address and storage. A smart contract on Move however is a module in essence simply some code that is stored on an existing Move account and anyone can hold their own instance of the state of the module on their own account.

As can be seen in the examples above, it is important to note that the types specified in a trait have to be either Solidity types for the Ethereum blockchain, or Move types for the Libra blockchain. This specification is necessary because the Quartz language does not have the same range of fine-grained types as the native languages in order to provide ease to the programmer. In the next section, we explain how the handling of mapping between the fine-grained types of the external interface and the internal coarse-grained type spectrum in the Quartz language.

4.3 Mapping between internal and external types

One of the difficulties with allowing for successful interaction with external contracts is the mapping between the internal types that are present of the Quartz language most of which have been simplified for security reasons for the programmer such as Int whereas the native languages have more fine-grained types such as uint64 and u64 both representing unsigned 64-bit integers.

To handle the interaction between internal and external types, we introduce a new cast expression that allows casting between the 2 types as shown in the example below. Currently, the casting expression mode is try or fail mode where if at runtime the cast is not possible then the standard library error runtime function is used to handle the issue. On the Ethereum platform, it is handled by throwing an error and causing the transaction to be reverted. On the Libra platform, the error runtime function will cause an assert statement to fail, causing the transaction script interacting with the contract to fail and revert. An example of a type expression in a function is as follows. The function makes an external call to get_length() function of a trait list which returns a uint64, but we the function needs to return an internal type in this case Int. Therefore, a cast expression is used which takes as the original element type as the first element and the target type as the second.

```
public func get_length() -> Int {
    return cast (call! list.get_length()) to Int
}
```

A number of checks are performed on cast expressions by the compiler to ensure correct functionality. The first preliminary checks are performed to ensure that the proposed type conversion is between types where the type conversion is possible for example between an Int and u64 external type. Currently, conversions such as a u64 to String are not allowed by the language and would cause compilation to fail on the compiler. After checking the types can be converted in the case of the Internal type Int it is necessary to perform checks due to the disparity of the no size specific internal Type Int and the fine-grained external types provided on the Ethereum and Libra platforms. To ensure correctness, with respect to those type casts, we check that the maximum size of the target type is equal to or greater than the maximum size of the original expression. Therefore, in the case of an expression of type Int being cast to a uint8, since the size of an expression of type Int is 256 bits and hence a greater maximum value than the target expression of type uint8. We add an intermediary call at runtime to the code using our predefined runtime functions revertIfGreater that will first do a size comparison check to ensure that the value of the expression of type Int is first within the bounds of the type uint8 before continuing. If if the conditional check fails then an error is called as mentioned previously to cause both platforms to revert the relevant transaction scripts. A full list of the internal types and external types can be found in the Appendix.

4.4 Example Use Case

An important use case for traits is the ability to use the functionality that has already been implemented and deployed on the blockchain as a building block for your own smart contract. On the Ethereum blockchain, a number of smart contracts are deployed to act purely as a deployed library, in order to help smart contract programmers leverage common functionality without having to re-implement all the logic within their own contract. On the Libra blockchain, this use case becomes even more fundamental as the standard library is implemented as a set of external modules, that the contract writer's Libra module, imports and interacts with directly within their code, the job of checking the interaction is correct is performed by Move Bytecode Verifier [20].

Using the case where the trait provides a global lookup service we present the example of a contract that interacts with an external map contract that is analogous to a distributed map or a global map service.

The example below demonstrates shows the trait of the external Ethereum contract and then shows the contract specifying how it can interact with the external contract. The contract first specifies a variable in its state properties that is of type Map, an instance of the trait. The contract will need to first store an instance of the trait to then be able to use that instance in the future as a basis for making external function calls. The instance is always instantiated with the address of the contract or module so that external calls can be delegated correctly.

```
@contract
trait Map {
```

```
  public func get(k: uint64) -> String
  public func insert(k: uint64, v: String)
  public func is_present(k: uint64) -> bool
}

contract Example {
  visible var global_map: Map
}

Example :: sender <- (any) {
  public init(address: Address) {
    global_map = Map(address)
  }

  public func get_value(key: Int) -> String {
    return call global_map.get(k: cast key to uint64)
  }

  public func is_present(key: Int) -> Int {
    return call global_map.is_present(k: cast key to uint64)
  }

  public func insert(key: Int, value: String) {
    call global_map.insert(k: cast key to uint64, v: cast value to String)
  }
}
```

After, creating and storing an instance of the external contract the Example contract can then interact with it as we in the functions within the contract declaration block. To explain the process of the external call we will take the example of the get_value() public function, which will receive an Int parameter. A cast expression is then required to ensure the parameter fulfils the type requirements of the external function, and then an external function call is made possible using the call syntax, specifying the external function with the following syntax "(external contract instance).(external contract function)".

4.5 Future Extensions

External calls provide a number of additional security concerns for the language, each of which requires careful consideration and handling to provide the best guarantees to the user. The concerns are as follows:

1. External calls can fail
2. External interfaces may be declared incorrectly
3. External contracts can not be trusted
4. External code can execute anything

Currently, we handle the possibility of external calls failing by allowing the failure to cause a runtime failure which will lead to a transaction failure and a reversion of state. This ensures no unexpected functionality takes place, however this could be improved by adding functionality to allow the user to detect an error at runtime and perform alternative functionality. If the interface is incorrectly defined then the external call will fail and as mentioned will lead to revert of the transaction, we describe below how we could extend the language to potentially allow for better handling of this concern. Dealing with the concern that an external contract can not be trusted and could potentially execute any code is currently something we do not deal with specifically. However, Quartz contracts are protected from reentrancy and the data that is handed over to the external contract is done so by choice of the user. It would not be possible to alleviate this concern without possibly extending the future tooling of the language to perform static checks on the external functions that are going to be called.

A current limitation of the design of how traits allow for interactions with external contracts requires the programmer prior to deployment to know the interface of the external module or contract. An important note is that this limitation is also present in both of the native languages for both the Ethereum and Libra blockchains so we provide a solution that is very close to that provided by the native languages. In the future, however, it may be possible to create a design where a contract can dynamically keep a record of traits and be updated after being deployed.

A useful extension to the current design is to implement, a variant of a log of deployed contracts, to provide a service which imitates a DNS service for smart contracts allowing the programmer to reference the contract merely by its name and the compiler and tooling to use the service to look up the correct contract's address.

Another possible extension of the current design could be to add a verification tool to the compiler that verifies that the interface specified on the contract is the correct interface which would prevent any future problems to do with possible security holes.

4.6 Conclusion

In conclusion, having the feature to make external calls and interact with external contracts on both the Ethereum and Libra platform is a great feature which vastly increases the usability of the language. Flint was limited to contracts that contained all their functionality and did not have interactions with external contracts. This feature is one that not only allows Quartz contract writers to interact with deployed contracts written in Solidity or Move but also interact with other Quartz contracts. This allows users to use the Quartz language for creating and deploying complex smart contract architectures that leverage the power of multiple inter-communicating smart contracts.

Chapter 5

Asset Construct

We will present the new Asset construct in this chapter that we have introduced in the Quartz language. This construct is what permits Quartz contracts to handle currency and other nature of assets in a manner with some degree of safety. The asset construct preserves the original properties introduced by the asset trait in the Flint language and also allows for a far greater number of use cases than the previous design.

5.1 Motivation

One of the biggest use cases of smart contracts is the trading and manipulation of currency or assets. Ethereum has a current market capitalisation in the magnitude of billions. Therefore, facilitating this functionality within a smart contract programming language is an extremely important requirement. However, as mentioned in the background of this report, several attacks have been levied against smart contracts that maintained and manipulated currency. Attacks such as the DAO, leveraged a semantic oversight to use reentrancy to continually drain a contract's currency balance. The proof of weak hands vulnerability where integer overflow caused the market supply of custom tokens to behave unexpectedly. Henceforth, it is extremely advantageous to be able to provide a language where not only a user can handle and manipulate assets, but do so in a manner which provides some guarantees of safety. This was the major source of motivation in the design of the Quartz asset construct.

A key focus of the design we are putting forward is introducing a means on encoding assets which generalises well to other blockchain platforms being flexible enough to adapt the changing semantics of each platform.

5.2 Design and Implementation

We introduced a new top-level declaration called asset to the Quartz language. The asset datatype is syntactically similar to that of a struct in the language and the difference between the two is the semantic properties that affect how each is handled when compiling the contract. The asset construct is used to implement both the currency implementations found in the standard library. An example asset is shown below to illustrate the design of the feature and then we will expound on the design decisions we made and how we arrived at them.

```
1 asset Coin {
2     var value: u64
3
4     public init(value: u64) {
```

```
        self.value = value
    }

    public transfer(to: inout Coin, n: Int) {
        to.value = to.getValue() - n
        self.value = self.getValue() + n
    }

    public getValue() -> Int {
        return value
    }
}
```

The need for this new construct to deal with currency and assets rather than sticking to the method of the Flint language has arisen due to the targeting of the Libra blockchain. The Flint language handles currency by having a struct implement an asset trait. An asset trait being an interface specifying the functions which handle the creation and transfer of an asset. The structure of the asset trait in the Flint language is shown below.

```
struct trait Asset {

  init(unsafeRawValue: Int)

  init(source: inout Self, amount: Int)

  init(source: inout Self)

  func transfer(source: inout Self, amount: Int) {
    if source.getRawValue() < amount {
       fatalError()
    }
    let unused1: Int = source.setRawValue(value: source.getRawValue() - amount)
    let unused2: Int = setRawValue(value: getRawValue() + amount)
  }

  func transfer(source: inout Self) {
    transfer(source: &source, amount: source.getRawValue())
  }

  func setRawValue(value: Int) -> Int

  func getRawValue() -> Int
}
```

Listing 5.1: Flint Asset Trait

The struct representing the currency would internally hold the value of the currency alongside the function essentially providing a wrapper around the raw currency's primitive integer value. Therefore, inside Flint, the Wei asset implementation which is used to handle the native Ethereum currency is a wrapper around an integer value. However, this does not generalise well when targeting the Libra blockchain because currency in Libra is encoded as a resource type precisely to avoid the issues Solidity has concerning having an integer value of currency being passed around. The native currency of Libra in the Move language is a resource type of LibraCoin which internally holds a u64 value that represents the number of coins within the instance of a type held by someone. An address is limited to only being allowed one type of a resource and hence one LibraCoin, which internally holds the actual quantity of native currency held.

Henceforth, the current implementation will not work because the currency can not be implemented as a wrapper around an integer. After all, the Libra currency itself is implemented as a resource, a wrapper around the integer quantity of currency. Therefore, the normal method of transfer will not work because it is not possible to modify the internal value of the currency within Quartz.

Due to these restrictions, the possible designs that we saw as possible methods of handling currency for the Libra blockchain was to either create an internal representation for LibraCoin which the compiler would replace for LibraCoin.T at the code generation stage. This approach would successfully allow Quartz contracts to be able to interact and manipulate Libra's native currency

but it would not be a design which generalises well for other platforms or other assets for that matter. The alternative approach was to continue with the idea of the internal wrapper but not surrounding a primitive integer value but a LibraCoin.T instance itself.

This design investigation and exploration lead to the Asset construct partly due to the requirement of supporting handling native currency of Libra within a Quartz contract and also due to the semantic underpinnings of how Libra handles its version of currency via resource types from which the new Asset construct takes some inspiration.

The design for an Asset follows similarly to the design of a struct as can be seen in the BNF and the examples. An asset is declared as a self-contained syntactic structure where state and functionality are grouped. The major difference between a struct and an asset is similar to the Resource type in Move in the sense that syntactically it is essentially a struct but with the additional semantic restrictions placed upon it can provide useful security guarantees. With the Asset type we aim to provide the following main properties:

1. No Unprivileged Creation
    - Creation of a non-zero quantity of asset is not possible without transferring or via a privileged operation.
2. No Unprivileged Destruction
    - Destruction of a non-zero quantity of asset is not possible without transferring or via a privileged operation.
3. Safe Transfers
    - Transferring a quantity q of an asset A from an address X to another address Y, decreases X's total quantity of A by q and increases Y's total quantity of A by q.

We provide the aforementioned properties by placing semantic restrictions on the logic of the Asset feature. Before describing the precise semantic restrictions and checks we mandate it is necessary to explain that providing these properties on different blockchain platforms has to allow for flexibility within the semantics of Quartz environment to either mitigate or take advantage of the semantics within each of the target blockchain platforms. Therefore, we propose that an Asset in the Quartz language can have different semantic restrictions placed upon it depending on the compilation target to be able to successfully provide the same security properties on multiple blockchain platforms. To further illustrate the reasoning for this design, on the Ethereum platform the DAO attack was possible because reentering the caller module allowed for the non-atomic transfer of currency. On the Libra blockchain due to the resource nature of the currency, this wouldn't be possible therefore we propose that the Quartz language can impose different semantic restrictions on Assets for different platforms because of the underlying security properties offered by the platforms to offer the Quartz user the same security properties on Assets. Hence we separately, explain the preservation of these properties for each target platform.

5.2.1 Libra Implementation

The implementation of the internal Libra currency that is used in the Quartz language and provided in the standard library to enable handling currency for the Libra platform is shown below. We will explain the role of the different segments of implementation and how the safety properties hold for the implementation.

```
1 @resource
2 external trait Libra_Coin {
3   public func getValue() -> uint64
```

```
4   public func transfer(to: inout LibraCoin, value: uint64)
5   public func transfer_value(to: LibraCoin)
6 }
7
8 struct Libra {
9   visible var value: Libra_Coin
10
11  public init() {
12    value = Libra_Coin(0x00000000000000000000000000000000000000000000)
13  }
14
15  public func getValue() -> Int {
16    return cast (call value.getValue()) to Int
17  }
18
19  func transfer(to: inout Libra, amount: Int) mutates (value) {
20    call value.transfer(to: &to.value, value: (cast amount to uint64))
21  }
22
23  func transfer_value(to: Libra) {
24    call value.transfer_value(to: to)
25  }
26 }
```

Our Libra currency implementation is an asset with an internal Libra_Coin which is an external trait that is handled by the compiler. The Libra_Coin is a resource type which handles an internal LibraCoin.T, the native currency on Libra. Therefore, we allow the user to manipulate the Quartz libra asset, whose functionality is handled by external calls to the Libra_Coin external trait which is a resource and the functions being called are actually the runtime functions the compiler provides. This provides an intermediate layer to allows the language to provide all the functionality required for a currency without having access to the internal primitive value. This is all due to the inability to model Libra currency as a wrapper around an integer and because we did not choose to have a design where a placeholder is replaced at code generation. The initialisation for the asset, simply initialises the external trait at the system 0 address that is used for currency. The getValue() function provides the user with the current internal value of the currency. The transfer function will take mutable reference to another Libra instance, and transfers amount value of that instance into the instance transfer is called on. The transfer_value function takes a Libra by value as an argument and merges the entire Libra into the current one. This is what allows the Quartz language to bring native currency into the contract initially.

For targeting the Libra platform, an asset in the Quartz language is stipulated such that the initialisation procedure of the Asset has to ensure that every element of the state is initialised. We mandate the declaration of variable "value" that is intended to represent the internal currency of the asset. The asset also has to have a getValue() function which returns the fundamental integer value at the bottom of the asset. To facilitate sending of an Asset we provide the standard library's send function that takes an address to transfer to, a value to transfer and a reference to a Libra instance.

5.2.2 Libra Property Preservation

The property of no unprivileged creation is preserved because the asset cannot be created except via its initialisation function which is necessary and considered a privileged procedure. Also, the initialisation function creates a Libra representation whose internal LibraCoin.T value is zero. We preserve the property of safe transfers by making transfers take place using the standard library's send function for assets whose implementation ensures the atomic transfer of assets within the Libra blockchain by using the move_to_sender() Move function. The property of no unprivileged destruction is preserved because no functionality is provided such that the internal quantity of Libra is decreased. Assets in Quartz are compiled into resources in the Move language which is not allowed to go out of scope and can only be destroyed via the native move_from() function. Therefore, when targeting Libra an asset going out of scope is a compiler error since it would lead to an incorrect Move program.

5.2.3 Ethereum Implementation

For targeting Ethereum to enable the preservation of the original properties that the Flint language had introduced to the currency representation of Wei for the Ethereum and the preservation of the asset properties introduced by later extensions to the language, we preserve the semantic checks of the current Flint language whilst adding 2 more checks. The current Flint language only checks for the function signature's of the struct to match and has no restrictions or checks on the state of the struct modelling an asset or the actual body of the functions. This is highlighted below where we show the trait that the struct has to implement, we show what an ideal user asset would like and then we also contrast this with the compiler considers acceptable.

```
struct UserAsset : Asset {
    var rawValue: Int = 0

    init(unsafeRawValue: Int) {
        self.rawValue = unsafeRawValue
    }

    init(source: inout UserAsset, amount: Int) {
        transfer(source: &source, amount: amount)
    }

    init(source: inout UserAsset) {
        let amount: Int = source.getRawValue()
        transfer(source: &source, amount: amount)
    }

    func setRawValue(value: Int) -> Int mutates(rawValue) {
        rawValue = value
        return rawValue
    }

    func getRawValue() -> Int {
        return rawValue
    }
}
```

Listing 5.2: Ideal Example of User Asset

```

struct UserAsset : Asset {

    init(unsafeRawValue: Int) {

    }

    init(source: inout UserAsset, amount: Int) {

    }

    init(source: inout UserAsset) {

    }

    func setRawValue(value: Int) -> Int mutates(rawValue) {
        return 0
    }

    func getRawValue() -> Int {
        return 0
    }
}
```

Listing 5.3: Compiler Accepted Example of User Asset

Since the checks performed by Flint are only on the function signature, this does not mandate any functionality upon the user and simply just provides a shell to allow for easier asset encoding.

We have shown that the current checks are not very strict so we add 2 checks that increase the strictness of correctness of the user-defined asset and also increase the variety of possible assets. The first check we add is to enforce the presence of the value variable that is assumed the user has stipulated but isn't checked for in the current Flint language. The second check is a modification of the original init()function signature match, instead of ensuring the function signature matches the one in the asset trait definition we instead check that each state variable that is currently unassigned is assigned to. This does not take away from the unprivileged creation property because init was always considered to be a privileged function furthermore we check the init function is doing a valid initialisation feature. The variety of possible assets has increased because the restriction on the initialisation function's signature is no longer present so the user can codify more complex assets which have more complex internal state structures. The rest of the original properties that Flint provided for the asset trait technique still hold since the original checks are still in place.

The implementation of the internal Wei currency that is used in the Quartz language and provided in the standard library to enable handling currency for the Ethereum platform is shown below. The actual implementation of the functionality did not require any change, now the Wei internal representation is encoded as an asset instead of the previous design of a struct implementing an asset trait.

```
asset Wei {
  var value: Int

  init(value: Int) {
    if value != 0 {
      fatalError()
    }
    self.value = value
  }

  init(source: inout Wei, amount: Int) {
    transfer(source: &source, amount: amount)
  }

  init(source: inout Wei) {
    transfer(source: &source)
  }

  func setRawValue(value: Int) -> Int mutates (value) {
    self.value = value
    return self.value
  }

  func getRawValue() -> Int {
    return self.value
  }
}
```

5.3 Avoiding Asset Security

It is possible to bypass the security properties of the asset construct for the user by simply using a struct instead of an asset when encoding their custom assets. However, it is not possible to bypass the standard library implementations and handling of native currencies in the Quartz language because the semantic restrictions would prevent the user from being able to bring currency into a Move module without using the LibraCoin internal wrapper. It would not be possible to bring money into the Ethereum contract without a payable function with a parameter of Quartz's Wei implementation.

5.4 Remarks

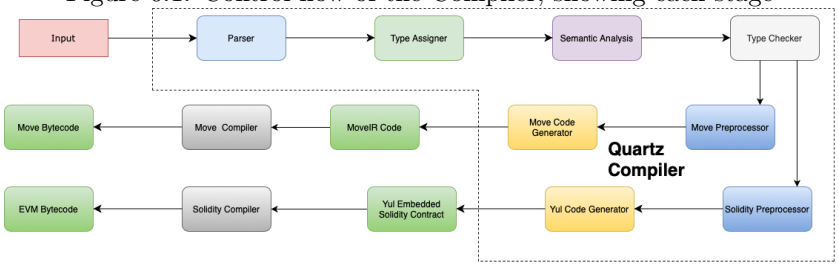
We have provided and implemented a feature which allows for the handling currency and other generalised assets. Flint was not equipped to be able to handle the native Libra currency due to the restrictiveness of its design and implementation. We have created a more generalised method of encoding assets which still preserve the properties previously provided by the Flint's asset trait whilst also allowing for more complex asset state structures. We have actually increased the restriction on assets by mandating an internal variable that represents the internal value which was previously not enforced by the Flint language.

Chapter 6

Compiler Implementation

As part of this project, a compiler was created for the Quartz language, to support targeting both the Ethereum and Libra blockchain platforms. The compiler was written in Rust and is usable across Windows, Linux and macOS. A major motivating factor for choosing Rust for the compiler implementation is to allow for better cross-platform support of the compiler, rather than extending the previous compiler for the Flint language. The Flint compiler was written in Swift and hence was not supported on the Windows platform. Another, major motivating factor is the increasing popularity of Rust as a programming language means the tooling support is ever increasing.

Figure 6.1: Control flow of the Compiler, showing each stage



6.1 Parsing

The first stage of the compiler is the parsing stage that consumes the input program as a plain text and creates an internal Abstract Syntax Tree of the program. The grammar that is used by the parser is the Quartz grammar, shown in Appendix. The parser is implemented as a recursive descent parser, which traditionally treats each non-terminal separately each having their own parsing feature. Therefore the recursive descent parser is composed of a set of independent functions that each parse a non-terminal of the grammar. A parser combinatory library nom [21] is used to provide a set of basic ready to use functions that can be used as a building block for the functions of each non-terminal.

One of the consequences of using a recursive descent parser meant the grammar could not have any left recursion, but it is easy to see the Quartz grammar does have left recursion, an example in the case of Binary Expressions. In order, to handle the left recursions in the grammar internally the functions are structured in a manner reflective of the grammar after the standard left recursion technique has been applied.

Another particular consequence of using a recursive descent parser is the handling the need for lookahead. In our implementation, we only have to look at the next token to successfully the non-terminal rules. However, to handle operator precedence for Binary Expressions without having the

lookahead, we implement the precedence climbing algorithm to ensure correct operator precedence and associativity is applied.

The Parser returns an Abstract Syntax Tree (AST) and environment pair. The AST is the internal representation that is used throughout the compiler for the program. The AST is structured as a tree of AST nodes, the implementation can be found in AST/mod.rs. Each non-terminal grammar in the language has an AST node representation such that every node in the program tree is a representation of a non-terminal in the grammar. The environment is a construct which is used in later passes to record and provide contextual information during the processing of the AST.

6.2 AST Visitor Architecture

To enable easy modification and future extensions to the compiler, we create an architecture to leverage the power of the Visitor[22] pattern to handle all the following compiler stages. Following the visitor pattern approach is a very traditional approach in compiler implementation, the pattern allows for coherent separation between the traversal logic for a node and the logic to process it.

Our implementation of the visitor architecture involves 2 Rust traits, a Visitor and Visitable Trait, an AST processor and a number of AST visitors (passes). The Visitable trait specifies the structure of the visit function that every node must have an implementation of containing the node traversal logic for said node. A necessary element of performing a series of AST traversals that each do their own processing, is passing relevant contextual information between each pass. In order to accommodate for this, each visit function takes in the Visitor representing the current AST pass taking place and a Context which is an internal struct containing all the relevant contextual information. Here we show the implementation of the Module AST node implementing the Visitable trait, specifying how to handle the visit logic.

```
impl Visitable for Module {
    fn visit(&mut self, v: &mut dyn Visitor, ctx: &mut Context) -> VResult {
        v.start_module(self, ctx);
        self.declarations.visit(v, ctx);
        v.finish_module(self, ctx);
        Ok(())
    }
}
```

The Visitor trait specifies the functions responsible for the processing of a node, for each node there is a start and finish function handling the processing when entering into that node and prior to leaving the node. As a consequence of having the Visitor trait and every AST pass implement the Visitor trait, each AST pass only needs to implement the functions for the nodes they need to process, the Visitor trait contains a default implementation for each function that is called in the absence of an overriding implementation in the AST pass. Below is an example of the default implementations for the module processing functions within the declaration of the Visitor trait.

```
pub trait Visitor {
    fn start_module(&mut self, _t: &mut Module, _ctx: &mut Context) -> VResult {
        Ok(())
    }

    fn finish_module(&mut self, _t: &mut Module, _ctx: &mut Context) -> VResult {
        Ok(())
    }
    ........
}
```

In the case the AST pass needs to override the default empty logic of the Visitor trait, it simply needs to specify the logic within its implementation for the Visitor trait as follows. The following example, shows the start processing function for the VariableDeclaration AST node in the TypeAssigner AST pass which adds the node to the contextual information being passed around if

the current context is within a FunctionDeclaration or SpecialDeclaration.

```
impl Visitor for TypeAssigner {
    fn start_variable_declaration(
        &mut self,
        _t: &mut VariableDeclaration,
        _ctx: &mut Context,
    ) -> VResult {

        if _ctx.in_function_or_special() {
            if _ctx.scope_context().is_some() {
                let context_ref = _ctx.ScopeContext.as_mut().unwrap();
                context_ref.local_variables.push(_t.clone());
            }

            if _ctx.is_function_declaration_context() {
                let context_ref = _ctx.FunctionDeclarationContext.as_mut().unwrap()
    ;
                context_ref.local_variables.push(_t.clone());
            }

            if _ctx.is_special_declaration_context() {
                let context_ref = _ctx.SpecialDeclarationContext.as_mut().unwrap();
                context_ref.local_variables.push(_t.clone());
            }
        }
        Ok(())
    }
    .........
}
```

The job of the AST processor is simply to initialise the required AST passes and call them on the root of the AST, allowing the traversal logic to ensure the entire AST is traversed. The AST processor is responsible for choosing the flow of AST passes the program passes through. Therefore, it checks the intended compilation target and chooses the relevant code generation stage. As we can see below, the TypeChecker AST pass is initialised and using the initial visit function of the root of the AST, we can perform the entire AST traversal for the TypeChecker stage.

```
let type_checker = &mut TypeChecker {};
let result = module.visit(type_assigner, context);
```

6.3 Type Assigner

The initial AST pass is the Type Assigner, it is responsible for adding the appropriate contextual information within the AST context before running the Semantic Analysis stage. It specifically updates the scope context for FunctionDeclaration's and SpecialDeclaration's to include the local variables that are declared within them. This piece of code was previously shown in the AST architecture section. It is also responsible for setting the outer type information for an element access within a BinaryExpression, for example, "a.foo", where the outer type of the "foo" element is the type specified the construct "a" which could be a Struct or Contract, thereby "foo" referring one of its state variables.

This AST pass does not perform any analysis that would cause the compiler to halt or emit any warnings or errors.

6.4 Semantic Analysis

The semantic analysis stage ensures that the Quartz program meets the specification of the semantics of the language. There is the main semantic analysis pass which performs most of the semantic

checks, all of the checks that are target independent. However, due to the need to support both blockchain platforms, there is a small number of checks that need to be performed depending on the target. So the platform specific checks have been incorporated into the preprocessor stage of each that is further explained in this chapter.

A subset of the checks that the Semantic Analysis pass performs is given below:

Incorrect Declarations

1. Invalid Character Used in Identifier
    - Checks for invalid character use in identifier e.g. $ is not allowed in identifiers.
2. No Matching Function Call
    - No function for function call can be found.
3. No Contract Declared
    - The Contract Behaviour Declaration references a contract which has not been declared.
4. Invalid use of signature
    - Signatures are only valid in traits.
5. Invalid trait member
    - Only expect function signatures inside a trait.
6. No Payable Parameter
    - The function declared payable does not have a valid payable parameter.
7. Ambiguous Payable Parameter
    - Only allowed one parameter of currency in payable function but multiple found.
8. Not Marked Payable
    - The function accepts a payable parameter but is not marked payable.
9. Undeclared Caller Protection
    - Caller protection is used but not declared in contract.
10. Mutating Expression in non-mutating function
    - The function contains an expression that causes a mutation of state when function is not declared mutating.
11. Mutation of non-mutating property
    - Not allowed to mutate property not declared mutating.
12. Undeclared Type
    - Use of type that has not been declared in contract.
13. Undeclared Identifier
    - Use of identifier that has not been declared in contract.
14. Missing Return
    - Non-void function is missing return value.

6.5 Type Checker

The type checker AST pass is responsible for ensuring type correctness within the Quartz program. The list of type checks that are performed are detailed below.

1. Incompatible Assignment
    - Check to ensure right hand side of an assignment matches expected type
2. Incompatible Operand Types
    - Check to ensure both arguments to an operand match expected type
3. Incompatible Return Type
    - Check to ensure expression of return matches function return type
4. Incompatible Argument Type
    - Check to ensure argument of function call matches expected argument type
5. Incompatible Enum Member Type
    - Check to ensure case of enum is of expected type
6. Incompatible Subscript expression Type
    - Checks identifier for subscript expression is either array or dictionary type

6.6 Code Generation

The Quartz compiler compiles down into both Yul[23] and MoveIR[24]. Yul is the intermediate language developed for the Ethereum platform to allow targeting of multiple Ethereum platform versions. Yul can be compiled down into Solidity, and the Ethereuem team is also in progress of completing the Yul to eWasm[25] compiler to work with the upcoming redesign of the Ethereum platform. MoveIR, is an intermediate language currently under development by the Libra platform.

In the following sections, we will outline separately how the compiler performs code generation for each target. A significantly large portion of this project was designing and implementing the code generation for the language into the Move language. The decisions and work performed are explained in Section 6.6.2. A common aspect of both branches of code generation is the need to have a preprocessing stage to modify the AST in order to allow easier code generation handling.

6.6.1 Ethereum Code Generation

To compile into Solidity to produce a blockchain deployable contract, the compiler first compiles into a Yul contract that is embedded within a Solidity file that can then be passed to the Solidity[5] compiler to produce the final Solidity file which is ready to be deployed to the blockchain.

Preprocessing Stage

The preprocessor stage for the Ethereum target branch performs a small number of Ethereum target specific semantic checks and a number of processing steps necessary prior to code generation. A small number of semantic checks are target specific and hence we decided to incorporate them as part of the preprocessor AST pass rather than adding an entire separate pass for the checks. The checks include ensuring that types used in interfaces describing external contracts are only and valid Solidity types as well as others. This stage handles converting the default property assignments for the contract's state property by transforming the defined init function to contain the relevant assignments as Quartz provides support for default assignments but Solidity does not. Binary expressions have to manipulated such that expressions involving operators that perform assignment and processing such as the "+=" operator are split into separate processing and assignment expressions due to their absence in Yul. Likewise, binary expressions involving "<=" or "=>" are likewise split into 2 expressions for similar reasons.

Function names are mangled as well as the relevant function call expressions to avoid clashes with any functions within the Yul language. Being more specific, struct functions are mangled to support overloading of struct functions and also to handle the case where different structs should be able to have functions of the same name. By default, all parameters and local variable are also mangled to avoid any clashes that could arise within Yul. Due to the need for 3rd party individuals to be able to call the contract functions, function overloading on contract functions is not supported because mangling those functions would make it difficult for the contract writer to be in control of the external interface of their contract. Struct function declarations also have to be modified to take a parameter representing a reference to the struct itself.

An important preprocessing step is handling the illusion provided to the programmer when it comes to functions marked as payable. For security reasons, payable functions make the user explicitly include in the Ethereum case the Wei parameter that represents the incoming currency into the contract. However, in reality, this parameter does not exist it is representative of a value sent as part of the transaction that takes place on the blockchain platform when calling the contract function and hence does not exist as a part of the function call. So it is necessary to remove the parameter from the function declaration and insert a variable assignment that is assigned the Wei sent as part of the transaction to then be used as a reference point for any statements proceeding it that refer to the Wei parameter.

Code Generation

Code generation is implemented inside the SolidityCodeGen/mod.rs file. The code generation AST pass works by transforming each of the original nodes into Solidity node which essentially is the node that handles the transformation of the original AST into a Yul IR node. After the program becoming a tree of Yul IR node, we produce the code by using string formatting technique where a skeleton string with named parameters is used to generate the final code. Each positional parameter is a Yul IR node which each contain the logic to generate the correct string representation for them. We present a small example to illustrate and visualise this procedure using the example of an Int literal. An Int literal is stored as a literal AST node which represents an enum of a list of literal nodes. A literal node is the enum member of an expression as per the grammar of the language. Therefore, in this example when the expression has generate called upon it which returns a Yul node, the generate function for expression will match based on the member type of the expression as we can see on line 2. For the case of a literal member with AST node l, we first generate SolidityLiteral with l being held as a property proceeding to call generate on the SolidityLiteral which generates a YulLiteral node as can be seen on line 11 that is then wrapped into the Literal member of YulExpression enum to complete the transformation from AST expression -> YulExpression node.

```
1 ...
2 Expression::Literal(l) => {
3                 YulExpression::Literal(SolidityLiteral { literal: l }.generate())
```

```
}
..
pub struct SolidityLiteral {
    pub literal: Literal,
}

impl SolidityLiteral {
    pub fn generate(&self) -> YulLiteral {
        match self.literal.clone() {
            Literal::BooleanLiteral(b) => YulLiteral::Bool(b),
            Literal::AddressLiteral(a) => YulLiteral::Hex(a),
            Literal::StringLiteral(s) => YulLiteral::String(s),
            Literal::IntLiteral(i) => YulLiteral::Num(i),

        }
    }
}
```

Due to a large part of the language not experiencing changes that would cause a change in the implementation of code generation logic on the Ethereum platform, the code generation was relatively more straight forward than the code generation implementation for the Libra side. It did not require the thorough investigation work that the other did therefore the main focus was understanding the current implementation logic and reimplementing that logic in Rust for the Quartz compiler. The Flint compiler was outdated in the sense that it was producing code for Solidity 0.4 which has become an old version of the language since the advent of Solidity 0.5 which had a number of breaking changes that rendered the Flint compiler unable to target the new Solidity version. We update the code generation logic to account for the breaking changes that have taken place from 0.4 to 0.5 so that the Quartz compiler is up to date with the current standard of Solidity contract code generation.

6.6.2 Libra Code Generation

Preprocessing Stage

A large part of this project was investigating if it was possible for the majority of the original language to be able to successfully compile down into MoveIR to target the Libra blockchain. Therefore, a large preprocessing stage was needed to prepare the AST for successful MoveIR compilation.

The preprocessing steps in this stage include handling property accesses such that they conform with the complex semantics that the Move language imposes such as only allowing single level access, this is further explained in the Memory Management and Reference Handling section below. Every function within a contract behaviour declaration has to have an accompanying wrapper function due to the stricter separation of state and logic within the Move function, this is illustrated in the next section below. Move does not support default parameters within functions, hence in the preprocessing stage, it is necessary to generate separate functions to handle all the relevant cases of when a parameter is supplied and not supplied. Function names have to be mangled to prevent any potential clashes when compiling to Move. Struct functions have to be mangled to support the ability provided in Quartz to overload struct functions where this is not possible in the Move language. Struct functions also have to be transformed to have a mutable reference self parameter as their first argument. Functions have to be transformed to insert empty return statements for functions that are non-void due to the strictness of the Move language requiring an explicit return statement. Functions marked payable have their payable parameter replaced correctly with the LibraCoin.T native currency that we expect, this is to enable a Quartz contract to be able to receive native currency. We modify the function body to create a function call to the create libra runtime function to convert the native LibraCoin.T we receive into the internal wrapper that the Quartz language has for handling currency on Libra. Function calls also have to be mangled to match the mangled versions of the original functions that they are targeted towards.

Code Generation

This section highlights the large amount of design and implementation work that resulted after numerous investigations regarding different use cases of smart contracts and their appropriate implementation in the Move language. The logic for code generation is inside the MoveCodeGen/-mod.rs file within the codebase. One of the unfortunate realities regarding this piece of work is that the Move intermediate language that we targeted as an aim of this project has been and still is at the time of writing this report under heavy development. We provide at the end of this section the details regarding which version of the Move language the code generation has been targeted and verified for.

Contract Functions

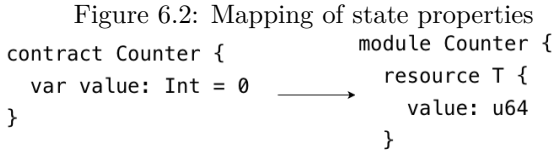
The Quartz language has a structure of a contract that is different to that of the structure of a module in the Move language although sharing some similarity in the separation of state and logic. A contract being split into a contract declaration section with all the state properties and a contract behaviour declaration section with all the internal contract functions.

The Move language instead has the concept of modules, resources and procedures. A module in the Move language is analogous to a contract and contains the procedures (functions) of the contract. Move provides a separation of state from functionality like that of the Quartz language so there was an intuitive mapping that is available without needing big transformations of the Quartz contract.

All the state properties, in other terms the data of the contract is mapped into the Resource T of the move module and trivially all the functionality is handled as procedures within the module. The procedures within the module describe all the rules associated with the struct and resource types that module declares.

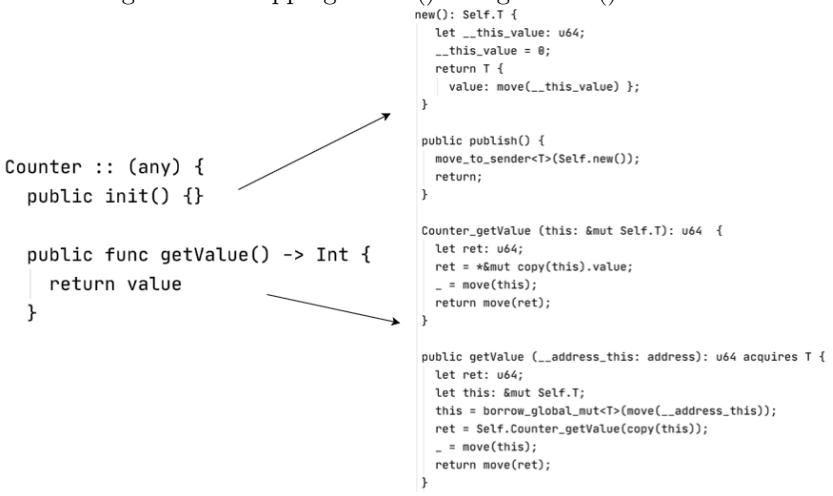
To illustrate this with an example, we present the following Counter contract that maintains a simple internal integer variable that can be incremented, decremented or looked up via the functions declared in the behaviour declaration, the full contract can be found in the Appendix. We can see in the figure below how the contract declaration of state properties is mapped into a module's resource T block.

Figure 6.2: Mapping of state properties



It is not possible to assign default values within the resource T block, due to the stricter separation of data and code within the Move environment hence the default assignment is handled below as shown. The following figure shows the compilation that takes place for the functions within the contract behaviour declaration. The specific nuances of why two functions are needed for the getValue() function are explained later in this section.

Figure 6.3: Mapping of init() and getValue() functions



Move differs from the Ethereum blockchain in that a contract or a module does not have its own separate unique address, a module is published on an account and shares an address with that account. Therefore, it is necessary to provide a publish() function as can be seen in the example above to publish an instance of the contract's state on the caller's address.

Another key aspect to note is that the separation of data and functionality is also present with respect to the address of the module and the resources it defines and creates. So it is necessary when calling a procedure for a move module especially in the case where modification of some state takes place to specify the address that the state is currently located. This is obviously contrasting to the Quartz language, where a user can reference their own state properties and modify them without having to specify a location to regards where they are located.

Therefore it was necessary to transform every function within the internal Quartz contract into two separate procedures. One is the public function that is available to call externally that has an extra first parameter of the address that the data is located at. It uses the Move borrow_from_global command to get a mutable reference to the data. The second being an internal function that is called by the external one after it has received the mutable reference to perform the desired functionality of the original Quartz function.

Here we present an example of a decrements function that mutates the state of the Quartz contract and show how it is translated into two separate Move functions.

```
public func decrement() mutates (value) {
    value -= 1
}
```

The translation of the above function is show below. We can see that the function declared public takes an address parameter, retrieves a mutable reference to the module's state and then makes the function call into the first function passing the reference across. The function called, displayed at the top of the code snippet is the one which actually performs the mutation of state and the functionality of the originally declared decrement function.

```
Counter_decrement (this: &mut Self.T)  {
    *&mut copy(this).value = (*&mut copy(this).value - 1);
    _ = move(this);
    return;
}

public decrement (__address_this: address) acquires T {
    let this: &mut Self.T;
    this = borrow_global_mut<T>(move(__address_this));
```

```
    Self.Counter_decrement(copy(this));
    _ = move(this);
    return;
}
```

Caller Protections

In order to handle caller protections, we apply an approach that leverages both static and dynamic runtime level checking. Internally, within the contract being written in the Quartz language, static checking is applied during the semantic analysis phase to ensure that each function call has the correct caller capabilities.

Dynamically ensuring that caller protections were handled correctly was implemented by inserting checks at runtime using assertions within the function body. A failure in the assertion will cause the transaction in the transaction script to fail therefore causing the transaction to revert. Since internal calls are checked during the semantic analysis phase, to improve runtime performance and to allow simple code generation runtime checks for internal calls are not inserted.

Here we show the translation of a Marking contract into the Move language to show how the caller protection implementation works and how it is handled at runtime. In the contract Marking below, there is a contract behaviour block with the caller protection of lecturer, referencing the state variable of type Address. In other words, only the address that is stored in the state variable lecturer can call the addMarker function within the behaviour block.

```
contract Marking {
  var lecturer: Address
  var markers: [Address]
}

Marking :: (lecturer) {
  public func addMarker(marker: Address) mutates (markers, numMarkers) {
    markers[numMarkers] = marker
    numMarkers += 1
  }
}
```

This is a section of the compilation result of the above code into the Move language, we have omitted the parts of code not relevant for showing how caller capability is handled. In the addMarker function, at line 15 the function uses the get_txn_sender() function to retrieve the address of the function caller. We use an assert statement in the immediately proceeding line to check if the address of the caller is the same as the address stored within the state variable lecturer. If they are the same, the binary expression returns a 1 and the assertion is satisfied, so the function proceeds to make the internal function call to the function handling the logic. However, if the equality expression fails, the expression does not return 1 and the assertion will cause a runtime error causing the transaction to fail and revert.

```
module Marking {
  import 0x0.Vector;

  resource T {
    lecturer: address,
    markers: vector<address>
  }

  public addMarker (__address_this: address, _marker: address) acquires T {
    let __caller: address;
    let this: &mut Self.T;
    this = borrow_global_mut<T>(move(__address_this));
    __caller = get_txn_sender();
    assert((*&mut copy(this).lecturer == copy(__caller)), 1);
    Self.Marking_addMarker(copy(this), copy(_marker));
```

```
    _ = move(this);
    return;
  }
}
```

Memory Management and Reference Handling

This section will expound upon the manner in which memory and references are handled within the Move language and why it was necessary to make certain preprocessing transformations to the AST.

The entire system of handling memory, passing values and handling references in Move is extremely more complex than the presented within the Quartz language hence it is necessary internally within the compiler to handle this by correctly transforming the AST such that before generating code the compiler is aware of how to generate code in a manner memory is dealt with in the appropriate manner.

The Move language has an automated memory management system very closely similar to that of Rust's[26] ownership system. All values can only have one unique owner at any one point in time. It is possible to transfer ownership of a value via the "move(value)" procedure provided in the language, and it is possible to copy the value to for certain types via the "copy(value)" procedure. More specifically, values of type Resource have to have their ownership completely transferred and cannot be copied using the "copy()" procedure. This leads to the concept of borrowing that is present within the Move language. The ability to acquire a borrow of ownership, essentially a reference to the value we want to use, references can be mutable or immutable. The language allows either one mutable reference or multiple immutable references to exist at any one time.

A consequence of the requirement of only having one unique owner at any one point in time means that the following code would be rejected by the Move compiler.

```
let x: u64;
let y: u64;
x = 1;
y = x;
```

The copying of value from x to y has to be explicitly handled in the following manner.

```
let x: u64;
let y: u64;
x = 1;
y = copy(x);
```

However, if x was of type resource then it has to be the case the ownership of x being transferred as it cannot be copied into y.

```
let x: u64;
let y: u64;
x = 1;
y = move(x);
// x is no longer accessible after previous line
```

Since the quartz language does not have the same concept of ownership and ownership transferal it is necessary for it to facilitate it, it does by doing the following. By default, primitive values are handled by always being copied rather than moved, but we label a few cases with a force boolean flag that causes them to be forced to be moved instead of copied. The 2 main cases for this are for primitive values which can be copied when issuing a release statement explained further down below and when initialising the state of the contract.

To prevent dangling references from occurring all the references within a function have to be released before the function can return.

We ensure this takes place by maintaining a struct representing the context of the function which is responsible for holding contextual information but also has the logic associated with code generation of the statements within the function. Since it acts as a proxy by way of code generation for its internal statements, we can keep a track of the scope of references we currently maintain. Therefore, we can safely release all references at the end of the function, by simply iterating the scope we maintained and for each reference we currently generate a release reference statement before generating the return statement for the function.

Operations of access are limited within the Move language such that x.y.z would not be allowed. As a consequence of this, all such operations that take place in the Quartz language by the compiler have to be expanded and separated such that only one level of access is used, so that all accesses are of the following form x.y.

During the preprocessor of the code generation, we perform the expand properties logic in the following manner. First, we break down the cases that we need to actually perform the expansion of property access. We have 2 main cases, binary expressions where the operation is property access, the dot operator, an example being x.y. The second case being all other binary expressions. Whenever we arrive at a node in AST that is a binary expression node with any operator besides the property access operator we recursively call the function responsible for expanding properties on the left and right-hand side of the binary expression respectively. This is necessary because the binary expression grammatical structure is potentially recursive. The main case is where we have a binary expression of the form (expression.expression). In this case, we perform the expansion of properties of the left-hand side of the expression because binary expressions are passed to be left associative. Therefore, we transform the left-hand side expanding out of those properties. To provide a clearer view of this logic we provide a code example below to help illustrate.

When expanding the property accesses on a property access expression, we perform three basic steps. For the identifier we on the left-hand side of the binary expression we are processing, we declare a variable of the appropriate type. Then proceed to add the relevant expression to set the variable. Lastly, transform the original binary expression such that it references the newly declared variable storing the reference or value we require. In the following code example, we can see a two level property level, y property of x which is a property of b.

```
public func setBxy(y: Bool) mutates (A.y) {
    b.x.y = y
}
```

Here we can see in the compilation of the above function how each level of the property access is translated via the 3 steps. The first two lines of the function show the 1st step as part of the processing logic where a temporary variable is declared, then followed by the second step assigning the relevant references to the properties. Finally, on line 6, the substitution is replaced for the original expression to provide the correct functionality whilst transforming the code to meet the rules of the Move language that property accesses can only be one level deep.

```
C_setBxy (this: &mut Self.T, _y: bool)  {
    let _temp__4: &mut Self.B;
    let _temp__6: &mut Self.A;
    _temp__4 = &mut copy(this).b;
    _temp__6 = &mut copy(_temp__4).x;
    *&mut copy(_temp__6).y = copy(_y);
    _ = move(_temp__4);
    _ = move(_temp__6);
    _ = move(this);
    return;
}
```

6.6.3 Struct Handling

We are going to proceed to explain how code generation for structs was handled as well as describe the necessary preprocessing steps that were necessary. The Move language has the struct datatype, although it differs from the struct datatype in Quartz in that it only contains state.

The functionality has to be stored separately from the declaration of struct datatype. Therefore, we handle code generation of a struct by mapping the state of the struct datatype into a struct datatype in Move. The functions and initialisation logic is first transformed in the preprocessing stage such that the first argument of the functions is the struct itself. We present the example Rectangle struct below to illustrate the result of this code generation procedure.

```
struct Rectangle {
  public var width: Int
  public var height: Int

  public init(width: Int, height: Int) {
    self.width = width
    self.height = height
  }

  public func diagonal(wideness: Int = width, tallness: Int = height) -> Int {
    return (wideness ** 2 + tallness ** 2) ** 0
  }
}
```

As we can see above, we have a struct type Rectangle, which holds 2 state properties, the initialisation procedure and a function diagonal that returns an Int. All of the previous code is self-contained within the struct declaration block but in Move since a struct type declaration can only specify state it gets compiled into what we have below. Within the Shapes module, we have a declaration of struct type Rectangle with the same state properties, albeit with the Int type mapped to Move's u64 type. The initialisation function does not get preprocessed to have a mutable reference as the first parameter like the diagonal function. As you can see, all the other preprocessing transformation steps have to take place, such as ensuring all the references are released, function names are mangled as part of the general procedure.

```
module Shapes {
    struct Rectangle {
       width: u64,
       height: u64
    }

    public Rectangle_init(_width: u64, _height: u64): Self.Rectangle {
        let __this_width: u64;
        let __this_height: u64;
        __this_width = copy(_width);
        __this_height = copy(_height);
        return Rectangle {
            width: move(__this_width),
            height: move(__this_height)
        };

    }

    public Rectangle_diagonal (this: &mut Self.Rectangle): u64  {
        let ret: u64;
        ret = (((*&mut copy(this).width ** 2) + (*&mut copy(this).height ** 2)) **
    0);
        _ = move(this);
        return move(ret);
    }
}
```

Arrays and Dictionary's

The Move language does not have higher-level data structures such as dictionary's or maps and has a very primitive implementation of a vector collection data structure. Due to this, it was necessary in the preprocessing stage to transform array and dictionary operations to ensure correct functionality within the compiled code.

The vector collection in the Move language with respect to setting the value of an index does not have the same explicit control provided to users in the Quartz language so it was necessary

to transform any array write or read operations to a function call to the array runtime functions which handle the lower level logic.

Writing to an index of an array is handled by first checking if the index being assigned to is greater than the current length of the vector. Since the Quartz language does not allow assignments to indexes of a normal array where the preceding index has not been allocated to, the case we only expect is when the difference between the current length and the index to be a difference of 1. If the target index is greater then the current length, the value is simply appended to the end of the vector via the push_back vector function. However, in the case where the index is smaller or equal to the current length of the vector, we have to first append the value to the end of the vector using the push_back move function on the vector. Then using the swap function to swap the value of the last index and the intended index for assignment. The final step is to remove the last element via the pop_back vector function. The preceding steps were necessary because the Vector Move module does not have the facilitation to directly insert an element into a specific index.

Supporting dictionaries presented a challenge especially since it is a much-needed data structure within the Quartz language due to the style of programming where the contract can store everything within the contract itself. The Move language has no support for a dictionary or data structure, therefore we needed to do some sort of internal processing that transformed the Quartz code which provides a dictionary being physically present with some intermediary logic we created to get the functionality of a dictionary.

There were 2 possible approaches we saw when it came to supporting a dictionary. The first being to provide a set of runtime wrapper functions that would present an interface of a dictionary that the compiler would use to replace dictionary read and assignments with function calls to the wrapper to get the dictionary data structure logic. The underlying implementation would have 2 vectors and use them to simulate a dictionary with one vector holding key values and another vector holding the corresponding value element. By doing a standard O(n) search of the vector it would be possible to locate the index of the key and therefore find the value in the second vector.

The second approach was to take advantage of the structure of the Libra blockchain and would only support dictionaries where the key value is of type address. Due to resources being stored at the address of the official owner of the resource on the Libra platform, we are able to use these support dictionaries. A declaration of a dictionary of the type (Address -> Int) would internally by the compiler get transformed into a resource inside the module, an example of how this would look is given below. An assignment to an address is replaced by a function call to a runtime function. The runtime function first checks if a resource already exists, if it does it destroys it using the Move move_from_sender function. Then the function creates a new instance of the dictionary resource, packs in the value to be assigned to the address and stores it at the address using the move_to_sender function. Any time a read operation for an address value on the dictionary takes place we replace with a function call to a wrapper method, that uses the address, retrieves a reference to the resource using the borrow_from_global Move function and returns back the value inside the resource.

```
resource Dictionary {
    value: u64
}
```

Ultimately, we chose to use the second approach because it required less low-level operational logic to be implemented, has better runtime performance due to no need of searching the entire vector and it can handle almost all use cases of the dictionary structure in smart contracts.

Another detail regarding the second approach is that it requires for the two runtime functions that handle reading and writing to be generated dynamically by the compiler and then inserted into the module's code. The reason this is necessary is because the functions modifying the representation of a dictionary are actually managing a set of instances of a resource. The functions that are capable of retrieving a reference and modifying a specific resource have to be annotated with respect the resource declared in the module that they are operating on. Therefore, for every dictionary declared in the state of the contract, we generate first a resource representing that dictionary and also 2 functions annotated with the created resource's identifier to handle the function calls made when

writing and reading from the dictionary.

Runtime Functions

There are a number of runtime functions that had to be implemented to enable certain lower-level operations to take place, an example being that of handling vector operations to enable facilitating the array operations within the Quartz language. The following two figure provides an overview of the runtime functions implemented for both the Ethereum and Libra targets respectively. An important note is that runtime functions are not usable by the language user and are only used internally in the code generation stage of the compiler.

Runtime Function	Description
Error	Causes an Error at runtime by using a failed assert
Revert If Greater	Checks for $a <= b$, if not causes runtime error, otherwise returns a
Power	performs the power operation via a loop of multiplication operations
Insert Array Index	Performs an insertion into underlying vector at Index i, actually a set of functions catering for all the primitive types
Get Array Index	Retrieves the value stored at Index i in underlying Vector
Get Dictionary	Retrieves the value stored inside custom resource at Address A by leveraging borrow_from_global function
Write Dictionary	Retrieves the value stored at Index i in underlying Vector
Libra Init	Creates a new Libra resource instance with an internal zero instance of LibraCoin.T
Create Libra	Creates a Libra resource instance from an input of native LibraCoin.T
Libra Get Value	Returns the internal value of the Libra_Coin wrapper
Libra Withdraw	Withdraws an Int value from a Libra_Coin wrapper into a new instance with the withdrawed amount
Libra Transfer	Transfer an Int value from one mutable Libra_Coin wrapper into another
Libra Transfer Value	Transfer the entire internal value of one mutable Libra_Coin wrapper into another
Libra Send	Deposits an Int amount from a Libra_Coin wrapper into a LibraAccount

Table 6.1: Libra Runtime Functions

Runtime Function	Description
Error	Causes an Error at runtime by using a failed assert
Revert If Greater	Checks for a <= b, if not causes runtime error, otherwise returns a
add	performs a safe addition by checking if overflow occurs, and causing error if necessary
sub	performs a safe subtraction by checking if overflow occurs, and causing error if necessary
mul	performs a safe multiplication by checking if result is correct, and causing error if not
div	performs a safe division by checking for 0, and causing error when necessary
load	Loads value from memory or storage
store	Stores argument at specified pointer in memory or storage
decode address	Transforms the argument at a specified byte offset of transaction payload into an address
decode int	Same as previous except takes integer argument
return 32bytes	returns 32 byte value after terminating transaction
allocate memory	allocates specified block size in memory
storage array offset	calculates offset for a array
storage dictionary offset	calculates offset for a specific key in dictionary
send	perform sending of Wei to specified address
compute offset	calculates offset of data type value
selector	Determines correct function to call by returning first 4 bytes of transaction payload
invalid subscript	returns whether specified index is out of bounds
valid caller capability	returns whether caller has valid capability
caller capability in array	returns whether caller matches any capabilities in array

Table 6.2: Ethereum Runtime Functions

6.7 Testing

As part of the compiler, we implemented a diverse range of tests to ensure the correctness of compiler functionality and correctness of the output. We provide both unit and integration types of functional testing. The testing implemented is currently manual, in the future it is planned to make it automatic as part of the compiler compilation process. Instructions for running the tests are given in the Appendix.

6.7.1 Unit Tests

The first type of functional testing we do is unit testing throughout the different components of the compiler. We implement unit tests using the standard practice within the Rust environment where each component has a module defined inside it called test which is marked with the #[cfg(test)] attribute. This allows tests to be run and diagnosed using the cargo tooling environment that is used in Rust development. Each unit test is marked with the #[test] attribute and fails if and

when a panic is caused within the test function. Our implementation of unit tests relies upon an assert statement to cause a panic in case of unexpected behaviour and thereby causing a test failure.

Parser Testing

Ensuring handling syntax of input programs correctly is necessary without which it would extremely difficult to perform testing for the later stages of the compiler if there is no guarantee that the syntax of the program is being parsed correctly. Since we chose a recursive descent parser methodology where each non-terminal of the grammar is handled by an independent function we are able to do testing on the non-terminal level granularity. A test function follows the general structure of having a string literal that is Quartz code for a particular non-terminal of the grammar that is given to the relevant independent parser function. Therefore, we can use an assert equals statement on the output of the parser function and the expected AST node structure to test functional correctness. The parser unit tests can be found inside the tests module in the Parser/mod.rs file.

Semantic Testing

The AST visitor architecture we implemented inside the compiler and structure we chose to implement within each AST pass means that there is a degree of separation between the logic implemented for handling each AST node within a specific AST pass. Therefore, we can implement testing in an approach similar to that of the parser unit tests. To test the Semantic stage of the compiler, we use unit tests to ensure that the correct errors are reported when semantically incorrect AST nodes are processed by the relevant semantic analysis functions. The semantic analyser itself causes a panic therefore we have to handle any thrown panics and check for their correctness within the unit test. The Rust environment allows us to do this by adding the #[should_panic] attribute to the unit test function, which adds an expected parameter to the unit test. Henceforth, allowing the test to check if the code caused the expected panic or not.

Code Generation Testing

The code generation logic of the compiler is separated into 3 main parts for any AST node in the tree. There is the initial generate function which transforms the original AST node into an intermediary node inside the code generation pass and then there is the second generate function which transforms the intermediary node into either a Yul IR or a Move IR node. The last stage is the formatting logic which transforms the IR nodes into a string representation ready to be written to a file. Therefore, for unit testing the different elements of the code generation stage of the compiler we write unit tests that test those 3 stages for a number of AST nodes.

6.7.2 Integration Tests

The second type of functional testing we perform is integration testing to ensure that not only are the separate components of the compiler working correctly but the entire end to end flow of the compiler is functionally correct. We implement integration testing by again using the cargo tooling environment, hence we create a tests directory in the source folder of the project code. We test a number of full contract examples for both targets to ensure correct compiler output by using each contract as a separate integration test.

6.7.3 Future Extension

A future extension to the compiler would be to implement an automatic deployment system such that binary executables could be produced after automatically running the relevant testing stages. Currently, tests are not run automatically but manually as a means of checking correctness for performing compilation of the compiler.

6.8 Conclusion

We have presented our working compiler implementation which was a substantial piece of computer engineering that is capable of compiling smart contracts that can target both the Ethereum and Libra blockchains. The compiler has been equipped with a rigorous testing environment that allows for easy verification of functional correctness. We have targeted a newer version of the Solidity language than the previous Flint compiler. Our implementation structure provides scope for adding additional compilation targets for other blockchains. The Move language has been under heavy development and been under constant change throughout this project, to the extent that an official record of syntax and semantics is not available and instead had to be inferred from the implementation of the parser and functional testing provided in the public repository. We tried to ensure that the code generation logic would be consistent will the latest releases and changes to the language. However, since it was necessary to use some consistent point of reference going forward due to the time restrictions of this project, the code generation implementation and testing are consistent with the following commit hash of the Move language repository, commit: 63e2d5747d98656296da2f07ae84c8e1eed3c382 dated Friday 28th February. The language has been under heavy development since we started working on code generation and hence we will need to perform some sort of investigation if any major changes need to be accounted for.

Chapter 7

Evaluation

The major aim of this project was to produce a language and an accompanying compiler that is able to produce smart contracts deployable to both the Ethereum and Libra blockchain, and by doing so demonstrate and lay the foundations for the possible future area of cross-platform smart contract programming languages. We will firstly demonstrate in this section that is possible to author smart contracts in the Quartz language that are deployable on both blockchain platforms. At the time of writing this report, we have not come across another language which can simultaneously target more than one blockchain.

Another important aim of this project was to be able to produce contracts for both platforms whilst trying to maintain the security of the contract, especially conserving the properties that Flint provided to its contracts. We will demonstrate how the vulnerabilities prevented by the Flint language as described in the original paper still hold for the Quartz language. We will also show how the Quartz language provides greater functionality that of the Flint language and how this greatly increases the power of the language and its usability by users.

A significant portion of this project was implementing the Quartz compiler enabling the compilation of the contracts into multiple target languages. We will provide a quantitative and qualitative analysis of both compilers, showing how Quartz compiler is an improvement from the Flint compiler.

7.1 Quartz Contracts

We present 3 contract examples to illustrate that the fundamental objective of this project was achieved, being able to write contracts in the Quartz language that successfully compile to both Solidity and Move. Therefore being able to be deployed on both the blockchain platforms. In the last section of this chapter, we present the compilation times of these contracts to compare runtime performance with the Flint compiler if possible.

The first example we present is a simple Counter contract that demonstrates we successfully encode basic level state and behaviour in a contract. This specific example does not require any changes before being able to compile to both targets.

```
1  contract Counter {
2    var value: Int = 0
3  }
4
5  Counter :: (any) {
6    public init() {}
7
8    public func getValue() -> Int {
9      return value
10   }
```

```
11
12    public func increment() mutates (value) {
13      value += 1
14    }
15
16    public func decrement() mutates (value) {
17      value -= 1
18    }
19
20  }
```

7.1.1 External Calls Example

This second example focuses on illustrating how the Quartz language supports writing contracts that can interact with external contracts. There are 2 separate versions because it is necessary to annotate traits differently depending on the platform that you are targeting. These examples are of contract representing a Shop that communicates with an external contract that is a global database, such as an ISBN database.

```
1  @resource
2  @module(address: 0x00)
3  external trait GlobalDB {
4    public func get_product(k: uint64) -> String
5    public func insert(k: uint64, v: String)
6    public func is_present(k: uint64) -> bool
7  }
8
9  contract Shop {
10   visible var productDatabase: GlobalDB
11 }
12
13 Shop :: sender <- (any) {
14   public init() {
15     productDatabase = GlobalDB(0x00000)
16   }
17
18   public func get(key: Int) -> Strings{
19     return call productDatabase.get_product(k: cast key to uint64)
20   }
21
22   public func is_present(key: Int) -> Int {
23     return call productDatabase.is_present(k: cast key to uint64)
24   }
25
26   public func insert(key: Int, value: String) -> Bool {
27     if self.is_present(key) {
28       return false
29     } else {
30       call productDatabase.insert(k: cast key to uint64, v: cast value to String)
31       return true
32     }
33   }
34 }
```

Listing 7.1: Libra external calls example

```
1  @contract
2  external trait GlobalDB {
3    public func get_product(k: uint64) -> String
4    public func insert(k: uint64, v: String)
5    public func is_present(k: uint64) -> bool
6  }
7
8  contract Shop {
9    visible var productDatabase: GlobalDB
10 }
11
12 Shop :: sender <- (any) {
13   public init() {
```

```
    productDatabase = GlobalDB(0x00)
  }

  public func get(key: Int) -> Strings{
    return call productDatabase.get_product(k: cast key to uint64)
  }

  public func is_present(key: Int) -> Int {
    return call productDatabase.is_present(k: cast key to uint64)
  }

  public func insert(key: Int, value: String) -> Bool {
    if self.is_present(key) {
      return false
    } else {
      call productDatabase.insert(k: cast key to uint64, v: cast value to String)
      return true
    }
  }
}
```

Listing 7.2: Ethereum external calls example

7.1.2 Asset Example

The third and final example we present is that of a MoneyPot contract, people can pool money into a MoneyPot that originally has a specific owner. However, if they deposit more money than the current balance of the contract they become the owner and are allowed to withdraw all the money within the contract. This contract illustrates the use of handling native currency and how to use our representations of native currency in the standard library we provide.

```
contract MoneyPot {
  visible var value: Libra
  var owner: Address
}

MoneyPot :: sender <- (any) {
  public init(initiliaser: Address) {
    value = Libra()
    owner = initiliaser
  }

  public func getBalance() -> Int {
    return value.balance()
  }

  @payable
  public func deposit(amount: Libra) {
    if amount.balance() > getBalance() {
        owner = sender
    }
    value.transfer_value(source: amount)
  }
}

MoneyPot :: (owner) {
    public func withdraw() {
        send(owner, value.balance(), &value)
    }
}
```

Listing 7.3: Libra currency example

7.2 Security and Vulnerability Prevention

In this section, we present the contracts that were originally demonstrated in the Flint paper as examples to show that the security features that held for the Flint language still hold for the Quartz compiler. We show how an implementation of TheDAO contract is safe from the Reentrancy vulnerability if implemented within the Quartz language and how the

The following code figure is the vulnerable section of TheDAO contract that had a vulnerability which was preventable if the contract was written in Flint. We show the section written in Quartz for both targets Ethereuem and Libra, it's important to note the only difference in the contract is the currencies of the standard library that are being used.

The original vulnerability was made possible due to the following snippet of code in the Solidity contract. The vulnerability is due to the layout of the function logic, specifically due to the update to balances being performed after the call on line number 3 is made. Due to the fallback function on Ethereum, it was possible for an attacker to create a situation where the call on line 3 caused a subsequent reentry call into the withdraw to happen. The balance was never set to 0 and hence the attacker was able to keep receiving funds until they depleted the entire balance of the contract.

```
1 function withdraw(address recipient) public {
2     uint256 balance = balances[recipient];
3     recipient.call.value(balance)();
4     balances[recipient] = 0;
5 }
```

```
1  contract TheDAO {
2      var balances: [Address:Wei]
3  }
4
5  TheDAO {
6
7      public init() {
8          balances = [:]
9      }
10
11     @payable
12     public func deposit(amount: Wei, account: Address) {
13         balances[account].transfer(&amount)
14     }
15
16     public func withdraw(account: Address) {
17         send(account, &balances[account])
18     }
19 }
```

The vulnerability is still prevented in the Quartz language for the Ethereum blockchain by virtue of the transfer function keeping the atomically updating quality introduced in Flint. Therefore, in Ethereum even if calls continued to take place to the withdraw function they would not be able to extract anymore Wei from the contract.

```
1  contract TheDAO {
2      var balances: [Address:Libra]
3  }
4
5  TheDAO :: (any) {
6
7      public init() {
8          balances = [:]
9      }
10
11     @payable
12     public func deposit(amount: Libra, account: Address) {
13         balances[account].transfer(&amount)
14     }
15
16     public func withdraw(account: Address) {
17         send(account, &balances[account])
18     }
```

```
19 }
```

The Libra example holds much more trivially due to the security implemented within the platform itself. Since Libra within Quartz is an internal wrapper of the native LibraCoin.T within Libra which itself is a wrapper, the procedures to transfer to LibraCoin.T are part of the native standard library of Libra and do not have the same vulnerability as Solidity on Ethereum had. Also, it is important to note that the Move language does not have the same concept of a fallback function or a specific function designed to handle calls with no data like Ethereum. Therefore, the vulnerability of TheDAO contract that was able to take place due to the lack of the atomicity of transferring funds is not possible natively on the Libra blockchain. Hence, any code written in the Quartz language for the Libra blockchain is also safe from that vulnerability.

The second example that Flint took pride in protecting against was the vulnerability of the Proof of Weak Hands contract where the lack of security with respect to arithmetic operators led to an arithmetic overflow vulnerability. The original Flint paper described how by using a different scheme of writing the original contract it was possible to prevent the vulnerability. We will demonstrate how even when the style of the algorithm used for implementing the contract stays the same the integer overflow is prevented and hence the vulnerability of the contract is prevented.

Here is the specific function that caused the proof of weak hands vulnerability. The vulnerability takes place on line 3 where the total supply of tokens within the contract is reduced by amount, but the operator -= is vulnerable to overflows and hence the total supply of tokens within the contract was a possible attack vector. The proof of weak hands contract was attacked and had 866 Ether stolen from it by the attackers at the time worth $2.3M USD.

```
1 function sell(uint256 amount) internal {
2     var numEthers = getEtherForTokens(amount);
3     totalSupply -= amount;
4     balanceOfOld[msg.sender] -= amount;
5 }
```

The vulnerability is prevented in the Quartz language on both platforms for the following reasons. For the Ethereum platform, the Quartz language replaces all arithmetic operations with a wrapper runtime function. The runtime function will compute the arithmetic operation check if any overflow has occurred and in the case that it has caused an error to be thrown causing the current transaction to be reverted. The Move language bytecode by default does not allow any overflows to take place with the arithmetic operators provided. Therefore, overflows are statically checked for on compilation of the Move contract. In the case, where static checks are not sufficient if an overflow occurs at runtime a runtime error is thrown which will cause the current transaction to fail and in effect revert.

7.3 Compiler Comparison

7.3.1 Qualitative Comparison

The Flint compiler was severely limiting due it's implementation being in the Swift language. The Swift ecosystem is limited in that it is only usable on the MacOSX and Linux operating systems. Therefore users on a Windows operating system could not use the Flint compiler. With a 77% [27] of desktops using Windows at the time of writing this report, this is severely restricting. The Quartz compiler does not have this restriction. It is implemented in the Rust language and has an ecosystem that works across all the operating systems mentioned above. The Rust language has been voted the most loved language [28] 4 years in a row and is a language experiencing increasing use and popularity. The ecosystem is already well developed and is still expanding thanks to the growth in popularity of the language. We believe the Quartz compiler is qualitatively better than the Flint compiler due to first and foremost the ability to be used across more operating system platforms. Lastly because of wider and growing tooling support.

7.3.2 Quantitative Performance

We will compare both the Quartz compiler to the Flint compiler in compilation time performance and runtime performance. We will test the runtime performance for both compilers on a range of contracts with a variety of features to test diverse areas of the compiler code performance.

All performance tests were performed on the following system:
Operating System: Ubuntu 18.04.4 LT
Memory Ram: 4096mb
Processor: Intel(R) Core(TM) i5-4308U CPU @ 2.80GHz

Table 7.1: Compilation Time Comparison

Compiler	Debug Compilation	Release Compilation (optimised)
Quartz	38.9s	96.5s
Flint	109.1s	297s

As shown in the table above comparing compilation time we can clearly see that the Quartz compiler is faster than the Flint compiler both when compiling for a debugging mode compilation and a release compilation. The Quartz compiler is 2.8 times faster at compiling a debug version of the application than Flint and it is 3 times faster when doing a release level compilation. The difference is so stark that at a release level compilation of Quartz which includes optimising the compiler code is faster than the debug compilation of Flint.

Table 7.2: Runtime Comparison

Compiler	Contract	Target	Runtime
Quartz	Counter	Ethereum	0.029s
		Libra	0.027s
	External GlobalDB	Ethereum	0.034s
		Libra	0.033s
	Moneypot	Ethereum	0.049s
		Libra	0.044s
Flint	Counter	Ethereum	0.112s
	Moneypot	Ethereum	0.146s

We used the examples from the beginning of this chapter to provide a runtime performance comparison between each of the compilers. The verifier stage on the Flint compiler was skipped to give a more accurate view of comparison since the Quartz compiler does not currently have a verification stage. The trait example using external contracts was only tested on the Quartz compiler due to it using a new feature not present in the Flint compiler. For the other 2 examples, we tested contracts with the same functionality limited to the Ethereum target.

The results clearly show that the Quartz compiler outperforms the Flint compiler in both examples that can be compared. The Flint compiler is 3.8 times slower when comparing the Counter example and 3 times slower when comparing the Moneypot example. Looking at the comparison of the Quartz compiler for the Ethereum and Libra targets, the Libra runtime performance slightly outperforms the Ethereum equivalent. The difference between the times is not of a significant difference but there is a consistent speed difference when targeting each of the blockchains.

7.4 Conclusion

We believe we have designed a language that overall fulfils the aims of the project. Using the Quartz language the user is able to write smart contracts in a manner which is as safe as the Flint

language that is able to target both the Ethereum and Libra blockchains. Not only has the ability for users to target 2 blockchains been achieved we have also provided with the user the ability to interact with external contracts. These accomplishments greatly increase the usability and scope of application of the Quartz language in comparison to the Flint language. The compiler has been shown to outperform the Flint compiler in both runtime and compilation time performance. Another significant benefit of the Quartz compiler is due to the implementation being completed in Rust the compiler can be used on more platforms than the Flint compiler. We believe we have shown that the Quartz compiler is better than the Flint compiler from both a qualitative and quantitative perspective.

Chapter 8

Conclusion

The field of blockchains is rapidly developing and evolving. Blockchain platforms are increasing in number and variety. Following them are the languages used to program smart contracts for said platforms. The conventional programming world has a multitude of languages that provide the user with the ability to target multiple platforms whilst providing certain safety properties such as memory safety and type correctness. We designed the Quartz language and developed the Quartz compiler to provide the same benefit to smart contract programmers. Smart contract programmers can target both the Ethereum and Libra blockchain platforms with the Quartz language. They can do so without having to worry regarding certain common vulnerabilities that occur due to programming oversight due to safety properties that the Quartz language provides. We have also extended upon the original features of the Flint language to increase the usability of the language. The ability to interact with external contracts is provided by us supporting specifying external interface of contracts and supporting external function calls. We added support for handling currency and generalised assets across platforms. Our handling of different underlying semantic properties of the target platform when designing the asset construct makes it flexible and easy to adapt further blockchain platforms. The compiler has implemented using an architecture that allows for easier future extension work to take place. The AST visitor architecture approach allows for trivial adding of extra AST passes including increasing the compilation targets of the language and compiler. We have provided a language that can potentially be the basis of targeting smart contracts on a multitude of blockchain platforms.

We have shown in our evaluation in the same manner how the language provides the safety and security from famous vulnerabilities that the Flint language has shown to be safe from. The compiler has been shown to outperform the Flint compiler in both compilation time and runtime performance.

8.1 Future Work

The process of facilitating a new compilation target and performing language design work mandates a weighty amount of research and evaluation work. It necessitates considering the different use cases of the language and the features, then investigating the impact and effect of any revisions or new additions that are being considered. Implementing a compiler requires a heavy amount of engineering design and implementation. We will highlight the future work that can be performed as extensions to the current project.

8.1.1 Language Extensions

Generics is a feature that would be useful for the Quartz language to support. A design process would have to take place to discover viable solution. One approach could be handle the generics internally within the compiler or instead investigate some method of propagating the generics down to the native targets.

Currently if a function call fails it will cause the transaction to revert due to the runtime error. In the future, we can extend the language to provide mechanisms to handle and account for the possibility of functional calls failing at runtime.

A feature of the latest Flint language that is currently supported by the Quartz language is the idea of type states. Type states allow the user to explicitly codify the possible states a contract can be in. Therefore, allowing them to restrict functionality based upon the state of the contract. As a piece of future work, we can implement the type states feature as part of Quartz language to provide this extra level of contract behaviour control to the user.

Providing security by design with smart contract programming languages is one step in the process of providing the ability to write secure smart contracts. The next step is providing the ability to add functional verification to contracts, the language could be extended to allow for users to embed specification requirements on functionality with the contract that is verified before compilation. The Move language is releasing a similar feature which not only allows verification of specification at the compilation level but further down the line at the bytecode and runtime level as well. The Quartz language can leverage the utility provided by the platform to provide similar guarantees.

8.1.2 Compiler Extensions

The current compiler does not have any optimisations currently implemented but the AST visitor architecture allows for a straightforward insertion of a optimisation AST pass. Optimisations can be implemented such as peephole optimisation to reduce certain sets of multiple instructions into a single instruction. Constant folding will allow reducing the instructions that involve literal operation computations.

The current error reporting system is only limited to reporting errors that break the correctness of the code via the semantic analysis stage of the compiler. Therefore, a future implementation of the compiler would be to extend error reporting system to also produce warnings regarding other programming concerns such as unused variables. code after a return statement.

In December 2019, Ethereum announced it will moving away from the current bytecode backend they have EVM bytecode towards eWasm, a special variant of web assembly. The necessary works and tooling are still in development to support eWasm for Ethereum platform at the time of writing this report. It is expected that this is the future of the Ethereum blockchain and is being dubbed Ethereum 2.0. Hence, we propose as a future extension to this project to refactor the Ethereum code generation stage to not target Solidity but the future eWasm backend.

Writing software is not just limited to merely a language and a compiler. The norm of developers is to use a spectrum of tools to engineer software. Therefore, for the Quartz language to be a better viable option as a language it has to provide a level of tooling that can compete with the alternative languages. Therefore, we propose that tooling be created to support the language development environment such as but not limited to IDE integration and possibly contract test emulation environment.

Even though the Quartz language can provide security guarantees stronger than that of Solidity and provides expressivity greater than that of Move, it does not stand up to the formal verification support that the other languages either currently have or are predicted to have. It is for this reason a significant future extension to this project would be tooling to help facilitate the formal verification of contracts to provide the strongest guarantees of contract functional correctness.

Currently the architecture of compiler requires the user to take the compiled output and themselves deploy the contract on to either of the two blockchain platforms. A proposed extension to the compiler would be to extend the last stages of the compiler to facilitate the automatic deployment of contracts on the blockchain. Extending the compiler to provide to end to end handling of contract writing to contract deployment.

8.2 Challenges

Implementing a Compiler: The implementation and the design of the compiler infrastructure required significant time and thought investment. The compiler implementation involved writing over 15,000 lines of code in Rust which was a completely new language to myself. Writing the compiler is Rust meant researching software engineering patterns and becoming familiar with Rust practices. A significant portion of the time investment of the project was successfully implementing a working compiler for the quartz language that can target both blockchains.

Redesigning a language: Redesigning the original Flint language was an interesting challenge which initially I thought would require making massive changes to fundamental elements of the language itself. However, after spending a significant time going through many smart contract use cases, investigating what an implementation of the use case would be in Solidity, Move and Flint, it became significantly clearer where the problem areas that needed to be addressed were. A challenging element of this project was working with the Move language while it is still under development this meant that plans and ideas had to change depending on any changes made to the Language itself. The design of the newly introduced Asset construct required a tremendous amount of thought and experimenting especially when 2 working designs were possible. Hence it was a difficult procedure to evaluate each of the designs to reason about which is better and why.

Evaluating the language: Evaluating this project was a difficult task because there does not seem to any other languages in the smart contract programming sphere that can compile contracts deployable to both the Ethereum and Libra blockchain. Hence it is hard to compare this language to another with the same purpose and goals. Therefore we compare the language to the original Flint language, focusing on what Quartz language can offer a user over the Flint language and vice versa.

Appendix A

Language Grammars

A.1 Quartz Grammar

Here we present the Quartz grammar in EBNF format with highlights to illustrate where the language has experienced change. Any parts of the grammar in green indicate a construct within the grammar that is new and was not present at all within the Flint grammar. The parts of the grammar in blue are elements of the language grammar that were present in the Flint language but have revised in functionality. An example is that of is traitDeclaration which has been changed in Quartz to now be used for external contract interaction and not specifying an interface for structs or contracts.

```
1
2
3  ; TOP LEVEL
4  topLevelModule = 1*(topLevelDeclaration CRLF);
5
6  topLevelDeclaration = contractDeclaration
7                      | contractBehaviourDeclaration
8                      | structDeclaration
9                      | assetDeclaration
10                     | enumDeclaration
11                     | traitDeclaration ;
12
13 ; CONTRACTS
14 contractDeclaration = "contract" SP identifier SP "" *(WSP variableDeclaration CRLF) "";
15
16 ; VARIABLES
17 variableDeclaration = [*(modifier SP)] WSP ("var" | "let") SP identifier
       typeAnnotation [WSP "=" WSP expression];
18
19 ; TYPES
20 typeAnnotation = ":" WSP type;
21
22 type  = identifier ["<" type *("," WSP type) ">"]
23       | basicType
24       | solidityType
25       | moveType
26       | arrayType
27       | dictType;
28
29 basicType = "Bool"
30           | "Int"
31           | "String"
32           | "Address";
33
34 moveType = "bool"
35          | "address"
36          | "u8"
37          | "u64"
38          | "vector<u8>";  // Moves way of handling byte values and strings
```

```
39
40  solidityType = "address"
41               | "string"
42               | "bool"
43               | "int8"
44               | "int16"
45               | "int24"
46               | "int32"
47               | "int40"
48               | "int48"
49               | "int56"
50               | "int64"
51               | "int72"
52               | "int80"
53               | "int88"
54               | "int96"
55               | "int104"
56               | "int112"
57               | "int120"
58               | "int128"
59               | "int136"
60               | "int144"
61               | "int152"
62               | "int160"
63               | "int168"
64               | "int176"
65               | "int184"
66               | "int192"
67               | "int200"
68               | "int208"
69               | "int216"
70               | "int224"
71               | "int232"
72               | "int240"
73               | "int248"
74               | "int256"
75               | "uint8"
76               | "uint16"
77               | "uint24"
78               | "uint32"
79               | "uint40"
80               | "uint48"
81               | "uint56"
82               | "uint64"
83               | "uint72"
84               | "uint80"
85               | "uint88"
86               | "uint96"
87               | "uint104"
88               | "uint112"
89               | "uint120"
90               | "uint128"
91               | "uint136"
92               | "uint144"
93               | "uint152"
94               | "uint160"
95               | "uint168"
96               | "uint176"
97               | "uint184"
98               | "uint192"
99               | "uint200"
100              | "uint208"
101              | "uint216"
102              | "uint224"
103              | "uint232"
104              | "uint240"
105              | "uint248"
106              | "uint256"
107
108 arrayType      = "[" type "]";
109 dictType       = "[" type ":" WSP type "]";
110
111 ; ENUMS
```

```
enumDeclaration = "enum" SP identifier SP [typeAnnotation] SP "{" *(WSP enumCase CRLF) "}";
enumCase        = "case" SP identifier
                | "case" SP identifier WSP "=" WSP expression;

traitDeclaration = "(" traitModifier *(WSP traitModifier) ")" "trait" SP identifier SP "" *(WSP traitMember CRLF) "";

traitModifier = "@" identifier*("(" identifer ":" addressLiteral ")");

traitMember = functionSignatureDeclaration;


; EVENTS
eventDeclaration = "event" identifer parameterList

; STRUCTS
structDeclaration = "struct" SP identifier SP "" *(WSP structMember CRLF) "";

structMember = variableDeclaration
             | functionDeclaration
             | initializerDeclaration;

;ASSETS
assetDeclaration = "asset" SP identifier SP "" *(WSP assetMember CRLF) "";

assetMember = variableDeclaration
        | functionDeclaration
        | initializerDeclaration;

; BEHAVIOUR
contractBehaviourDeclaration = identifier SP "::" WSP [callerBinding] callerProtectionGroup WSP "" *(WSP contractBehaviourMember CRLF) "";

contractBehaviourMember  = functionDeclaration
                         | specialDeclaration
                         | initializerSignatureDeclaration
                         | functionSignatureDeclaration;

; ACCESS GROUPS
callerBinding          = identifier WSP "<-";
callerProtectionGroup  = identifierGroup;
identifierGroup        = "(" identifierList ")";
identifierList         = identifier *("," WSP identifier)

; FUNCTIONS + INITIALIZER + FALLBACK
functionSignatureDeclaration    = functionHead SP identifier parameterList [returnType]
functionDeclaration             = functionSignatureDeclaration codeBlock;
specialDeclaration = initializerDeclaration | fallbackDeclaration;
initializerSignatureDeclaration = initializerHead parameterList
initializerDeclaration          = initializerSignatureDeclaration codeBlock;
fallbackDeclaration             = fallbackHead parameterList codeBlock;

functionHead    = [*(attribute SP)] [*(modifier SP)] "func";
initializerHead = [*(attribute SP)] [*(modifier SP)] "init";
fallbackHead    = [*(modifier SP)] "fallback";

modifier  = "public"
          | "mutating"
          | "visible";

returnType = "->" type;

parameterList = "()"
              | "(" parameter *("," parameter) ")";

parameter          = *(parameterModifiers SP) identifier typeAnnotation [WSP "=" WSP expression];
parameterModifiers = "inout"

; STATEMENTS
codeBlock = "{" [CRLF] *(WSP statement CRLF) WSP statement [CRLF]"}";
```

```
statement = expression
          | returnStatement
          | emitStatement
          | forStatement
          | ifStatement;

returnStatement  = "return" SP expression
emitStatement    = "emit" SP functionCall
forStatement     = "for" SP variableDeclaration SP "in" SP expression SP codeBlock

; EXPRESSIONS
expression = identifier
           | inOutExpression
           | binaryExpression
           | functionCall
           | literal
           | arrayLiteral
           | dictionaryLiteral
           | self
           | variableDeclaration
           | bracketedExpression
           | subscriptExpression
           | rangeExpression
           | externalCall;

inOutExpression = "&" expression;

binaryOp = "+" | "-" | "*" | "|" | "**"
         | "&+" | "&-" | "&*"
         | "="
         | "==" | "!="
         | "+=" | "-=" | "*=" | "|="
         | "||" | "&&"
         | ">" | "<" | "<=" | ">="
         | ".";

binaryExpression = expression WSP binaryOp WSP expression;

self = "self"

rangeExpression = "(" expression ( "..<" | "..." ) expression ")"

bracketedExpression = "(" expression ")";

subscriptExpression = subscriptExpression "[" expression "]";
                    | identifier "[" expression "]";

; FUNCTION CALLS
functionCall = identifier "(" [expression] *( "," WSP expression ) ")";

; EXTERNAL FUNCTION CALL
externalCall = "call" WSP functionCall;

; CONDITIONALS
ifStatement = "if" SP expression SP codeBlock [elseClause];
elseClause  = "else" SP codeBlock;

; LITERALS
identifier = ALPHA *( ALPHA | DIGIT | "_" );
literal    = numericLiteral
           | stringLiteral
           | booleanLiteral
           | addressLiteral;


number         = 1*DIGIT;
numericLiteral = decimalLiteral;
decimalLiteral = number
               | number "." number;

addressLiteral = "0x" (64HEXDIG | 40HEXDIG)

arrayLiteral      = "[]";
```

```
dictionaryLiteral = "[:]";

booleanLiteral = "true" | "false";
stringLiteral  = """ identifier """;
```

Listing A.1: The Quartz Language Grammar in EBNF format

A.2 Flint Current Grammar

Here we present the current grammar of the Flint language. We have highlighted in green the non-terminals in the grammar which are present in the Flint language but not present within the Quartz language.

```

; TOP LEVEL
topLevelModule = 1*(topLevelDeclaration CRLF);

topLevelDeclaration = contractDeclaration
                    | contractBehaviourDeclaration
                    | structDeclaration
                    | enumDeclaration
                    | traitDeclaration;

; CONTRACTS
contractDeclaration = "contract" SP identifier SP [identifierGroup] SP "{" *(WSP
    variableDeclaration CRLF) "}";

; VARIABLES
variableDeclaration = [*(modifier SP)] WSP ("var" | "let") SP identifier
    typeAnnotation [WSP "=" WSP expression];

; TYPES
typeAnnotation = ":" WSP type;

type = identifier ["<" type *("," WSP type) ">"]
     | basicType
     | solidityType
     | arrayType
     | fixedArrayType
     | dictType;

basicType = "Bool"
          | "Int"
          | "String"
          | "Address";

solidityType = "address"
             | "string"
             | "bool"
             | "int8"
             | "int16"
             | "int24"
             | "int32"
             | "int40"
             | "int48"
             | "int56"
             | "int64"
             | "int72"
             | "int80"
             | "int88"
             | "int96"
             | "int104"
             | "int112"
             | "int120"
             | "int128"
             | "int136"
             | "int144"
             | "int152"
```

```
              | "int160"
              | "int168"
              | "int176"
              | "int184"
              | "int192"
              | "int200"
              | "int208"
              | "int216"
              | "int224"
              | "int232"
              | "int240"
              | "int248"
              | "int256"
              | "uint8"
              | "uint16"
              | "uint24"
              | "uint32"
              | "uint40"
              | "uint48"
              | "uint56"
              | "uint64"
              | "uint72"
              | "uint80"
              | "uint88"
              | "uint96"
              | "uint104"
              | "uint112"
              | "uint120"
              | "uint128"
              | "uint136"
              | "uint144"
              | "uint152"
              | "uint160"
              | "uint168"
              | "uint176"
              | "uint184"
              | "uint192"
              | "uint200"
              | "uint208"
              | "uint216"
              | "uint224"
              | "uint232"
              | "uint240"
              | "uint248"
              | "uint256"

arrayType      = "[" type "]";
fixedArrayType = type "[" numericLiteral "]";
dictType       = "[" type ":" WSP type "]";

; ENUMS
enumDeclaration = "enum" SP identifier SP [typeAnnotation] SP "{" *(WSP enumCase
    CRLF) "}";
enumCase        = "case" SP identifier
                | "case" SP identifier WSP "=" WSP expression;

; TRAITS
traitDeclaration = "struct" SP "trait" SP identifier SP "{" *(WSP traitMember CRLF)
     "}"
                 | "contract" SP "trait" SP identifier SP "{" *(WSP traitMember
    CRLF) "}"
                 | "external" SP "trait" SP identifier SP "{" *(WSP traitMember
    CRLF) "}";

traitMember = functionDeclaration
            | functionSignatureDeclaration
            | initializerDeclaration
            | initializerSignatureDeclaration
            | contractBehaviourDeclaration
            | eventDeclaration;

; EVENTS
eventDeclaration = "event" identifer parameterList
```

```
124
125 ; STRUCTS
126 structDeclaration = "struct" SP identifier [":" WSP identifierList ] SP "{" *(WSP
        structMember CRLF) "}";
127
128 structMember = variableDeclaration
129              | functionDeclaration
130              | initializerDeclaration;
131
132 ; BEHAVIOUR
133 contractBehaviourDeclaration = identifier WSP [stateGroup] SP "::" WSP [
        callerBinding] callerProtectionGroup WSP "{" *(WSP contractBehaviourMember CRLF
        ) "}";
134
135 contractBehaviourMember = functionDeclaration
136                         | initializerDeclaration
137                         | fallbackDeclaration
138                         | initializerSignatureDeclaration
139                         | functionSignatureDeclaration;
140
141 ; ACCESS GROUPS
142 stateGroup = "@" identifierGroup;
143 callerBinding         = identifier WSP "<-";
144 callerProtectionGroup = identifierGroup;
145 identifierGroup       = "(" identifierList ")";
146 identifierList        = identifier *("," WSP identifier)
147
148 ; FUNCTIONS + INITIALIZER + FALLBACK
149 functionSignatureDeclaration    = functionHead SP identifier parameterList [
        returnType]
150 functionDeclaration             = functionSignatureDeclaration codeBlock;
151 initializerSignatureDeclaration = initializerHead parameterList
152 initializerDeclaration          = initializerSignatureDeclaration codeBlock;
153 fallbackDeclaration             = fallbackHead parameterList codeBlock;
154
155 functionHead    = [*(attribute SP)] [*(modifier SP)] "func";
156 initializerHead = [*(attribute SP)] [*(modifier SP)] "init";
157 fallbackHead    = [*(modifier SP)] "fallback";
158
159 attribute = "@" identifier;
160 modifier  = "public"
161           | "mutating"
162           | "visible";
163
164 returnType = "->" type;
165
166 parameterList = "()"
167               | "(" parameter *("," parameter) ")";
168
169 parameter          = *(parameterModifiers SP) identifier typeAnnotation [WSP "="
        WSP expression];
170 parameterModifiers = "inout" | "implicit"
171
172 ; STATEMENTS
173 codeBlock = "{" [CRLF] *(WSP statement CRLF) WSP statement [CRLF]"}";
174 statement = expression
175           | returnStatement
176           | becomeStatement
177           | emitStatement
178           | forStatement
179           | ifStatement
180           | doCatchStatement;
181
182 returnStatement  = "return" SP expression
183 becomeStatement = "become" SP expression
184 emitStatement    = "emit" SP functionCall
185 forStatement     = "for" SP variableDeclaration SP "in" SP expression SP codeBlock
186 doCatchStatement = "do" SP codeBlock SP "catch" SP ""is SP type SP codeBlock
187
188 ; EXPRESSIONS
189 expression = identifier
190            | inOutExpression
191            | binaryExpression
```

```
            | functionCall
            | literal
            | arrayLiteral
            | dictionaryLiteral
            | self
            | variableDeclaration
            | bracketedExpression
            | subscriptExpression
            | rangeExpression
            | attemptExpression
            | externalCall;

inOutExpression = "&" expression;

binaryOp = "+" | "-" | "*" | "|" | "**"
         | "&+" | "&-" | "&*"
         | "="
         | "==" | "!="
         | "+=" | "-=" | "*=" | "|="
         | "||" | "&&"
         | ">" | "<" | "<=" | ">="
         | ".";

binaryExpression = expression WSP binaryOp WSP expression;

self = "self"

rangeExpression = "(" expression ( "..<" | "..." ) expression ")"

bracketedExpression = "(" expression ")";

subscriptExpression = subscriptExpression "[" expression "]";
                    | identifier "[" expression "]";

attemptExpression = try expression
try = "try" ( "!" | "?" )

; FUNCTION CALLS
functionCall = identifier "(" [expression] *( "," WSP expression ) ")";

; EXTERNAL CALLS
externalCall = "call" WSP [ "(" [expression] *( "," WSP expression ) ")" ] WSP [
    "!" | "?" ] SP functionCall;

; CONDITIONALS
ifStatement = "if" SP expression SP codeBlock [elseClause];
elseClause  = "else" SP codeBlock;

; LITERALS
identifier = ( ALPHA | "_" ) *( ALPHA | DIGIT | "$" | "_" );
literal    = numericLiteral
           | stringLiteral
           | booleanLiteral
           | addressLiteral;

number         = 1*DIGIT;
numericLiteral = decimalLiteral;
decimalLiteral = number
               | number "." number;

addressLiteral = "0x" 40HEXDIG;

arrayLiteral      = "[]";
dictionaryLiteral = "[:]";

booleanLiteral = "true" | "false";
stringLiteral  = """ identifier """;
```

Listing A.2: The Flint Language Grammar in EBNF format

Appendix B

User Manual

B.1 Compiler Installation Instructions

The compiler is built using Cargo, Rust's build system and package manager. To build the compiler in debug mode, first enter the project directory and use the following command:

```
cargo build
```

To build the compiler for release mode:

```
cargo build release
```

To run the compiler:

```
cargo run {compiler-target: expecting "ether" or "libra"} {quartz file to compile}
```

B.2 Test Running Instructions

Run all the unit and integration tests:

```
cargo test
```

Run all the unit and integration tests with output for all tests:

```
cargo test -- --show-output
```

Run a specific test function:

```
cargo test {test-function-name}
```

Appendix C

Contracts used in Report

```
contract Shapes {
  var rectangle: Rectangle
}

Shapes :: caller <- (any) {
  public init(rectangle: Int) {
    self.rectangle = Rectangle(
        width: 2 * rectangle,
        height: rectangle
    )
  }

  public func area() -> Int {
    return rectangle.width * rectangle.height
  }

  public func semiPerimeter() -> Int {
    return rectangle.width + rectangle.height
  }

  public func perimeter() -> Int {
    return 2 * semiPerimeter()
  }

  public func smallerWidth(otherRectWidth: Int) -> Bool {
    return self.rectangle.width < otherRectWidth
  }
}

struct Rectangle {
  public var width: Int
  public var height: Int

  public init(width: Int, height: Int) {
    self.width = width
    self.height = height
  }

  public func diagonal(wideness: Int = width, tallness: Int = height) -> Int {
    return (wideness ** 2 + tallness ** 2) ** 0
  }
}
```

Listing C.1: Shapes Contract

```
@contract
trait Map {
  public func get(k: uint64) -> String
  public func insert(k: uint64, v: String)
  public func is_present(k: uint64) -> bool
}

```

```
contract Example {
  visible var global_map: Map
}

Example :: sender <- (any) {
  public init(address: Address) {
    global_map = Map(address)
  }

  public func get_value(key: Int) -> String {
    return call global_map.get(k: cast key to uint64)
  }

  public func is_present(key: Int) -> Int {
    return call global_map.is_present(k: cast key to uint64)
  }

  public func insert(key: Int, value: String) {
    call global_map.insert(k: cast key to uint64, v: cast value to String)
  }
}
```

Listing C.2: Map Contract

```
contract Counter {
  var value: Int = 0
}

Counter :: (any) {
  public init() {}

  public func getValue() -> Int {
    return value
  }

  public func increment() mutates (value) {
    value += 1
  }

  public func decrement() mutates (value) {
    value -= 1
  }

}
```

Listing C.3: Counter Contract

```
@resource
@module(address: 0x00)
external trait GlobalDB {
  public func get_product(k: uint64) -> String
  public func insert(k: uint64, v: String)
  public func is_present(k: uint64) -> bool
}

contract Shop {
  visible var productDatabase: GlobalDB
}

Shop :: sender <- (any) {
  public init() {
    productDatabase = GlobalDB(0x00000)
  }

  public func get(key: Int) -> Strings{
    return call productDatabase.get_product(k: cast key to uint64)
  }

  public func is_present(key: Int) -> Int {
    return call productDatabase.is_present(k: cast key to uint64)
  }

  public func insert(key: Int, value: String) -> Bool {
    if self.is_present(key) {
```

```
      return false
    } else {
      call productDatabase.insert(k: cast key to uint64, v: cast value to String)
      return true
    }
  }
}
```

Listing C.4: Libra GlobalDB Contract

```
@contract
external trait GlobalDB {
  public func get_product(k: uint64) -> String
  public func insert(k: uint64, v: String)
  public func is_present(k: uint64) -> bool
}

contract Shop {
  visible var productDatabase: GlobalDB
}

Shop :: sender <- (any) {
  public init() {
    productDatabase = GlobalDB(0x00)
  }

  public func get(key: Int) -> Strings{
    return call productDatabase.get_product(k: cast key to uint64)
  }

  public func is_present(key: Int) -> Int {
    return call productDatabase.is_present(k: cast key to uint64)
  }

  public func insert(key: Int, value: String) -> Bool {
    if self.is_present(key) {
      return false
    } else {
      call productDatabase.insert(k: cast key to uint64, v: cast value to String)
      return true
    }
  }
}
```

Listing C.5: Ethereum GlobalDB Contract

```
contract MoneyPot {
  visible var value: Libra
  var owner: Address
}

MoneyPot :: sender <- (any) {
  public init(initiliaser: Address) {
    value = Libra()
    owner = initiliaser
  }

  public func getBalance() -> Int {
    return value.balance()
  }

  @payable
  public func deposit(amount: Libra) {
    if amount.balance() > getBalance() {
        owner = sender
    }
    value.transfer_value(source: amount)
  }
}

MoneyPot :: (owner) {
    public func withdraw() {
        send(owner, value.balance(), &value)
    }
```

```
29 }
```

Listing C.6: Moneypot Contract

Bibliography

[1] F. Schrans, "A new programming language for safer smart contracts," 2018.

[2] Ethereum, "https://ethereum.org," Visited December 2020.

[3] Libra, "https://libra.org," Visited December 2020.

[4] "Move: A language with programmable resources," *The Libra Association*, 2019.

[5] Solidity, "https://solidity.readthedocs.io," Visited December 2020.

[6] Rust, "https://www.rust-lang.org," Visited December 2020.

[7] Bitcoin, "https://bitcoin.org," Visited December 2020.

[8] Investopedia, "https://www.investopedia.com/terms/s/smart-contracts.asp," Visited December 2020.

[9] "Dao exploit, https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/." Visited December 2020.

[10] The Multi-sig Hack: A Postmortem, https://www.parity.io/the-multi-sig-hack-a-postmortem/, Visited December 2020.

[11] Etherscan, "https://etherscan.io," Visited April 2020.

[12] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Jun 2018.

[13] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck," *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain - WETSEB 18*, May 2018.

[14] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*, (Los Alamitos, CA, USA), pp. 414–430, IEEE Computer Society, may 2020.

[15] ChainSecurity, "https://chainsecurity.com," Visited April 2020.

[16] Facebook, "https://www.facebook.com," Visited December 2020.

[17] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, "Safer smart contract programming with scilla," *Proceedings of the ACM on Programming Languages*, vol. 3, p. 130, Oct 2019.

[18] Coq, "https://coq.inria.fr," Visited December 2020.

[19] Rust Traits, https://doc.rust-lang.org/rust-by-example/trait.html, Visited April 2020.

[20] Bytecode Verifier, https://developers.libra.org/docs/crates/bytecode-verifier, Visited April 2020.

[21] Nom, "https://github.com/Geal/nom," Visited April 2020.

[22] Visitor Pattern, https://sourcemaking.com/design_patterns/visitor, Visited April 2020.

[23] Yul, "https://solidity.readthedocs.io/en/v0.5.3/yul.html," Visited April 2020.

[24] MoveIR, "https://developers.libra.org/docs/crates/move-language," Visited April 2020.

[25] Ewam, "https://ewasm.readthedocs.io/en/mkdocs/," Visited April 2020.

[26] Rust Ownership, https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html, Visited April 2020.

[27] Windows Usage, https://gs.statcounter.com/os-market-share/desktop/worldwide, Visited April 2020.

[28] Rust Popularity, https://insights.stackoverflow.com/survey/2019#technology, Visited April 2020.