Efficiently transform contracts written in Peyton Jones Contract Descriptive language to Solidity

V. U. Wickramarachchi

Efficiently transform contracts written in Peyton Jones contract descriptive language to Solidity

V. U. Wickramarachchi
Index No: 14001543

Supervisor: Dr. Chamath Keppitiyagama
Co-Supervisor: Dr. Kasun Gunawardena

January 2019

Submitted in partial fulfillment of the requirements of the
B.Sc in Computer Science Final Year Project (SCS4124)

DECLARATION

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name: V. U. Wickramarachchi

……
Signature of Candidate Date:

This is to certify that this dissertation is based on the work of

Ms. V. U. Wickramarachchi

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Principle Supervisor's Name: Dr. Chamath Keppitiyagama

……
Signature of Principal Supervisor Date:

Co-Supervisor's Name: Dr. Kasun Gunawardena

……
Signature of Co-Supervisor Date:

ABSTRACT

Financial contracts play a major role in the modern economy. Due to a large variation of contracts being traded in financial markets, a standard representation for contracts was required in order to eliminate the ambiguity imposed by natural languages. Peyton Jones et al. catered this need by proposing a combinator library embedded in Haskell which enabled contract representation and valuation. However, every financial contract represented as such suffered from the same fundamental problem. If a contract is executable, the parties involved in the contract needed to trust a central counterparty to give them the correct results after execution. The interference of this middleman introduced certain risks as well as a significant amount of overhead.

In this dissertation, a novel approach to facilitate autonomous contract execution was proposed, exploiting the features and use cases of the Ethereum blockchain and its scripting language, Solidity. The approach involves transforming contracts written in the Peyton Jones' Contract Descriptive language to Solidity using a special purpose compiler. The result of this transformation is a smart contract equivalent to the traditional financial contract. The transformed smart contract is deployed and executed on the Ethereum blockchain using an Ethereum client.

The proposed solution was evaluated against existing attempts to design autonomous financial contracts. The research shows that a contract written in the Peyton Jones' Contract Descriptive language can be transformed to a smart contract which executes autonomously in a trustless environment. As a result, it was concluded that financial contracts could operate without a central counterparty with increased performance and reduced overheads.

PREFACE

Transformation of a financial contract to a smart contract is a novel approach proposed in this study. The objectives and aims of this study has not been explored by any other previous research of this particular domain. A novel design was introduced in order to facilitate this transformation. Two parts used in the design model; extending of the Peyton Jones' Contract Descriptive Language and the external contract (Marketplace) has been inspired by a parallel work done in this domain. Apart from that, the proposed design was solely my own work and a method similar has not been proposed in any other study relevant to this domain.

The implementation methodology used in order to build the compiler was proposed by myself. Even though a parallel work related to this concept was identified towards the latter part of this study, the objectives, aims, design and the implementation of that is drastically different from what is proposed in this study. The evaluation model used in this study falls in the line of a standard evaluation model of proof-of-concept. However, it was further improved by myself with the input of my supervisors.

ACKNOWLEDGEMENT

I would like to express my sincere appreciation to my principal supervisor, Dr. Chamath Keppitiyagama and co-supervisor Dr. Kasun Gunawardena for their constant guidance and encouragement, without which this work would not have been possible. For their unwavering support, I am truly grateful.

I would also like to extend my sincere gratitude to all the examiners and evaluators of my research for providing feedback on my research proposal and interim evaluation to improve my study.

My sincere thanks go out to our final year computer science project coordinator Dr. H. E. M. H. B. Ekanayake for his encouragement and support in keeping this research focused and on-track.

Foremost my special thanks to my parents for providing me a solid foundation in education and all the courage and love gave me on every moment. They are the guiding stars which strengthen me to become the person who I am today.

Finally, I express my sincere gratitude to all my friends who supported and encouraged me on all cause of challenges I faced during this research. All the help given by everyone to make this research a success owns my great appreciation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Acronyms

CDL	Contract Descriptive Language
DSL	Domain Specific Language
DSEL	Domain Specific Embedded Language
HDL	Hardware Description Language
HOL	Higher Order Logic
EBNF	Extended Backus-Naur Form
ANTLR	Another Tool for Language Recognition
NFA	Non-deterministic Finite Automata
EAC	Ethereum Alarm Clock
EVM	Ethereum Virtual Machine
IR	Intermediate Representation
AST	Abstract Syntax Tree

Chapter 1 - INTRODUCTION

1.1 BACKGROUND TO THE RESEARCH

With the rapid development of digitization of the world, everything from our own identities to the cars we drive, the businesses we run, the markets we operate in are all digitized and linked to the Internet. Technological advancement in such a scale will open many avenues for modern businesses, governments and financial markets. However, along with these advancements, an open question remains. If everything is being digitized and information is readily accessible by anyone anywhere, are today's centralized authorities the best way to look forward to the future?

1.1.1 FINANCIAL MARKETS AND CONTRACTS

A financial market is a market in which people trade financial securities and derivatives such as futures and options at low transaction costs. A financial market performs thousands of transactions per second. The New York Stock exchange is one such example of a financial market.

The adverse effects of centralized control are highlighted drastically in financial markets [1]. The digitization of financial markets started in the 1970's. Along with digitization, the trading volumes in these markets have increased massively. Trading in these markets happen through financial instruments such as financial derivatives (referred to as 'financial contracts' or 'contracts'). Derivatives are financial contracts, which derive their value off a spot price time-series, which is called 'the underlying'. The underlying asset can be equity, index, commodity, or any other asset.

Few examples of financial contracts are;

- Zero Coupon Bond - A bond that is issued at a deep discount to its face value but pays no interest.

- American Option - A put option or call option that can be exercised at any time on or before its expiration date.
- European Option - May be exercised only at the expiration date of the option, i.e. at a single pre-defined point in time

Traditional financial contracts require a third party for the purpose of executing the contract in addition to the parties involved in the agreement due to its centralized nature. The involvement of such third parties increases the security risk of exercising contracts because the result of the execution solely depends on the third party. The parties involved in the contract need to trust the third party to provide them the correct result.

The high-level goal of this research is to redefine the financial markets and financial contracts in a highly-automated, trustless, transparent environment.

The benefits of such an environment are as follows.

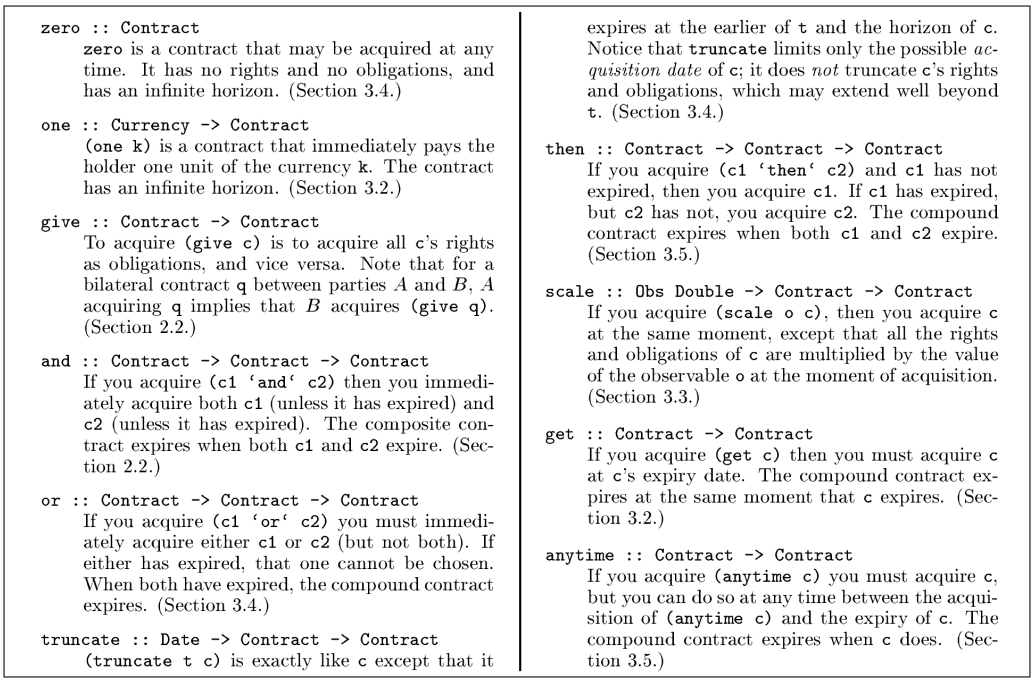
- High availability – With the increase in trading volumes, availability of the market is a critical component of financial trading. A trustless environment eliminates the single point-of-failure and reduces downtime of the market.
- Reliability – The approach used in this research to build a trustless environment for financial markets eliminates risks such as credit risks (The risk of failure of a counterparty to perform its obligation as per the contract), which as a result increases the reliability of the agreement.
- Efficiency – An automated process of executing contracts is much more efficient than a manual execution by a third party. Decreases contract execution overhead significantly.

In order to build such an environment, this research maps two existing domains; namely; The *Peyton Jones' Contract Descriptive language* and *Smart contracts for Ethereum*.

1.1.2 PEYTON JONES' CONTRACT DESCRIPTIVE LANGUAGE

Financial contracts portray a major role in the finance sector and the modern economy. The finance industry has an enormous vocabulary of jargon (options, swaps, futures, swaptions, etc.) to catalogue all typical combinations of such financial contracts. Due to the inherent ambiguity of natural language, it is considered unsuitable to express financial contracts. The financial domain lacked a universal domain specific language (DSL) to describe financial contracts in the past.

An influential paper by Simon Peyton Jones [2] is one of many attempts to create a DSL that would mitigate controversy and stimulate automated processing of complex contracts. It leverages ideas from functional programming such as Haskell and uses a precise set of primitive combinators to express financial agreements. A key feature of this notation is composability: new indefinitely complex contracts can be defined based on existing simpler ones. Due to their nested structure and composability, this DSL is well-suited for automated processing, including contract valuation and to describe unforeseen contracts. The usage of this has been proven through commercialization of the concept as well [3]. Figure 1.1 shows the combinators introduced by Peyton Jones' et al. to describe financial contracts.



`zero :: Contract`
`zero` is a contract that may be acquired at any time. It has no rights and no obligations, and has an infinite horizon. (Section 3.4.)

`one :: Currency -> Contract`
`(one k)` is a contract that immediately pays the holder one unit of the currency `k`. The contract has an infinite horizon. (Section 3.2.)

`give :: Contract -> Contract`
To acquire `(give c)` is to acquire all `c`'s rights as obligations, and vice versa. Note that for a bilateral contract `q` between parties A and B, A acquiring `q` implies that B acquires `(give q)`. (Section 2.2.)

`and :: Contract -> Contract -> Contract`
If you acquire `(c1 'and' c2)` then you immediately acquire both `c1` (unless it has expired) and `c2` (unless it has expired). The composite contract expires when both `c1` and `c2` expire. (Section 2.2.)

`or :: Contract -> Contract -> Contract`
If you acquire `(c1 'or' c2)` you must immediately acquire either `c1` or `c2` (but not both). If either has expired, that one cannot be chosen. When both have expired, the compound contract expires. (Section 3.4.)

`truncate :: Date -> Contract -> Contract`
`(truncate t c)` is exactly like `c` except that it expires at the earlier of `t` and the horizon of `c`. Notice that `truncate` limits only the possible *acquisition date* of `c`; it does *not* truncate `c`'s rights and obligations, which may extend well beyond `t`. (Section 3.4.)

`then :: Contract -> Contract -> Contract`
If you acquire `(c1 'then' c2)` and `c1` has not expired, then you acquire `c1`. If `c1` has expired, but `c2` has not, you acquire `c2`. The compound contract expires when both `c1` and `c2` expire. (Section 3.5.)

`scale :: Obs Double -> Contract -> Contract`
If you acquire `(scale o c)`, then you acquire `c` at the same moment, except that all the rights and obligations of `c` are multiplied by the value of the observable `o` at the moment of acquisition. (Section 3.3.)

`get :: Contract -> Contract`
If you acquire `(get c)` then you must acquire `c` at `c`'s expiry date. The compound contract expires at the same moment that `c` expires. (Section 3.2.)

`anytime :: Contract -> Contract`
If you acquire `(anytime c)` you must acquire `c`, but you can do so at any time between the acquisition of `(anytime c)` and the expiry of `c`. The compound contract expires when `c` does. (Section 3.5.)

Figure 1.1 – Combinators for defining contracts

1.1.3 Ethereum Blockchain and Smart Contracts

The introduction to Bitcoin [4] in 2009 led the world to a new era of decentralized technologies in finance and other industries. Out of these distributed platforms, the most interesting technology is Ethereum [5] for this study. Ethereum is a decentralized platform with mutually distrusting nodes, for running smart contracts.

Smart contracts are applications that run just the way they are programmed, with virtually zero possibility of any censorship, fraud, third party interference or downtime [6]. The idea of smart contracts, notably Ethereum, became the first practical implementation of the concept of autonomous contracts. The term *'smart contract'* evolved even before the emergence of blockchains [7]. One could even simply consider a vending machine as a primitive predecessor of a smart contract which enforces the agreement that a coin can be traded for i.e. a can of soda. However, the definition in concern for this study would be, "A program code that enforces agreements on the Ethereum blockchain among peers in the network." The significant feature of a smart contract is, once initiated, the participating parties cannot stop or reverse the transaction, unless allowed by the same or another smart contract. This immutability feature reduces security vulnerabilities that could affect financial contracts and its parties, enabling us to build the autonomous, trustless environment required. Since everyone in the Ethereum network could see how smart contracts execute and due to their immutable nature, each participant could independently verify that the system is fair.

Basically, smart contracts are programs stored and executed on an embedded virtual machine by participants in the Ethereum peer-to-peer network. Programs may perform any number of actions, such as updating its state, executing other programs or sending values to users or programs. Anyone can execute or deploy programs to the network for a small fee. The participants in the network continuously verify that they agree on the states of the deployed programs.

In the high-level languages currently used for Ethereum contract development, contracts are structured much like modules or classes in traditional programming languages. One such language is Solidity [8], the Turing-complete scripting language for Ethereum. Contracts are ultimately encoded as byte code for the Ethereum Virtual Machine (EVM) that is embedded in all clients in the network [9].

Ethereum has two main entities: *user accounts* and *contract accounts* that send messages between each other. If the recipient of a message is a contract, it will execute a bit of code, which might send new messages to other users or contracts. The contract accounts are controlled by the program which constitutes the contract. Contract accounts have access to a non-shared persistent memory, which they use to store their state [9]. Each account has exactly one automatically generated address, which is used as its unique identifier [9]. Additionally, all accounts have an *ether* balance. Ether is a transferable asset that is built into the Ethereum system.

1.1.4 Transactions and Execution Costs on Ethereum

The simplest kind of message in a transaction is to transfer a certain amount of ether from one account to another. If a contract receives a message, its code gets executed and a value may be returned to the sender. All transactions are executed by nodes participating in the network known as *verifiers*. Since the EVM is Turing-complete, there is a possibility that contracts would enter into infinite loops. In order to prevent this, users have to pay a certain fee at a fixed price for each execution step they trigger. This is done using *gas*, a resource bought using ether. Gas is necessary to execute contracts and the amount of gas supplied in the message limits the length of the computations it triggers. Before a transaction is made, the user specifies a gas price and a maximum limit of gas to be used for the transaction.

1.2 Research Problem and Research Questions

1.2.1 Research Problem

Financial contracts are a vital underpinning of the modern financial domain. With the ongoing technological advancement in the world, business and finance are increasingly being automated. As a result, the incentives to let computer programs interpret, enforce and execute contracts have also increased. There have already been several efforts to design formal languages and combinator libraries [2] [10] [11] to analyze or execute financial contracts. Hence, the idea of deterministic representation of contracts and reducing ambiguity is more common.

However, all of the existing implementations suffer from the same fundamental problem: if a financial contract is executable, the involved parties have to trust the executor to give them the correct result. Financial contracts are even exposed to a number of risks including the counterparty risk, where a certain party involved may opt out before the expiry date of the contract and credit risk, which is the loss that may occur from the failure of any party to abide by the terms and conditions of the financial contract.

In order to mitigate risks of a third-party involvement, this study proposes to have the contract execute itself, without any possibility of interference from an executor or other third parties. The solution involves mapping the traditional financial contract domain to the smart contract domain. As a result, the trustless environment required for contract execution is obtained. Further, the risks mentioned above are also eliminated as smart contracts feature autonomous execution and immutability, making it impossible for the parties involved in the contract to opt out of the agreement in an unconventional manner.

1.2.2 Research Question 01

Is it possible to facilitate an efficient transformation of a contract written in Peyton Jones' contract descriptive language to a self-executing smart contract in Solidity?

1.2.3 Research Question 02

Is it possible to preserve the properties of the Peyton Jones' language while accommodating the blockchain features in a transformed contract?

1.3 Research Aim and Objectives

The main aim of this research is to eliminate the dependence on third-parties to execute financial contracts and to facilitate autonomous execution of contracts in a trustless environment in order to reduce risks encountered when exercising financial contracts. This study focuses on achieving this by combining properties of the Peyton Jones' Contract Descriptive Language and the properties of the Ethereum blockchain such as immutability and decentralization.

The objectives of the research are as follows.

- Enhance financial contracts written in Peyton Jones' Contract Descriptive language, to embed properties required in the Ethereum blockchain domain
- Develop a source-to-source compiler to transform a Peyton Jones' financial contract to a smart contract
- Explore the advantages of transforming a Peyton Jones' financial contract to a smart contract
- Explore the ability of the transformed contract to act as the original contract
- Discover the abilities of the transformed contract to execute at a future date

1.4 Justification for the Research

Taking into consideration the importance of financial contracts to the economy, the motivation behind this study is to enhance the uses of them by automating its functions and eliminating the need of a middle man to execute the contracts. This idea is both possible and practical, as a result of emerging technologies such as smart contracts based on the Ethereum blockchain. As the blockchain technology is a trending area of research in the present context, accommodating the features of the blockchain to contracts will be a turning

point in the financial domain. The use of distributed ledger technology for the finance domain has been a topic of interest since the dawn of blockchains [3] and financial markets such as the London Stock Exchange have already begun discussions on how best to incorporate blockchain technologies to the financial domain, which validates the importance of the proposed solution in this study.

Most of the existing formal languages for financial contracts are functional in nature. Contracts defined using many of these languages are compositional where complex contracts are created by combining smaller contracts. It has been shown that contractual agreements are very well suited to be expressed in formal languages which are purely functional [12] over imperative languages.

Therefore, rather than re-writing financial contracts as smart contracts, it is more beneficial to efficiently transform them from the domain specific language (Peyton Jones' Contract Descriptive Language) to the scripting language (Solidity) of the Ethereum blockchain which is used to represent smart contracts. Thus, the best of both worlds is exploited; unambiguity and the composability of a concise declarative domain specific language (DSL) and trustless, autonomous execution of blockchain based smart contracts.

Researchers have successfully introduced DSLs in the past which could describe and evaluate smart contracts with specific operational semantics [2] [13] [14]. A functional language with an extensive type system (Idris) has also been introduced for safer development of smart contracts [15]. Further, approaches have been introduced where financial agreements could be securely managed using self-executing smart contracts. One such approach includes introducing a financial declarative DSL which is executed by the nodes of a blockchain network [16].

However, even if there are examples of re-implementations of financial contracts in the Ethereum blockchain platform, no evidence has been found in transforming the readily described financial contracts in one of the most stable financial declarative DSLs (Peyton Jones' Contract Descriptive Language) to the smart contracts.

The significance of the problem lies in the need to derive the best of both domains where the motive is to preserve the compositional nature and also to exploit the advantages of executing financial contracts in a trustless environment. The importance of the outcome of this research is pointed towards both the financial domain and the computer science research areas respective to blockchain technologies.

1.5 METHODOLOGY

As the first step of the research approach, the existing representation of financial contracts is refined in order to facilitate multiple representations of financial contracts. This is required in order to map the existing contracts to a different domain. The next step focuses on developing a source-to-source compiler to transform contracts written in Peyton Jones' Contract Descriptive language to Solidity. The approach followed for this step is discussed in detail in sections 3 and 4.

The final step focuses on devising a mechanism to evaluate the transformed contract, identify benefits of such a transformation and the extent to which the properties of the Peyton Jones' contract are preserved. This step would also include determining the methodology to embed properties specific to smart contracts with relevance to gas consumption, self-execution at a future date, etc. The problems faced with regards to gas usage and self-execution will be thoroughly analyzed and discussed. The complete analysis and discussion of this step is included in section 5.

Figure 1.2 represents a high-level diagram of the proposed research methodology.

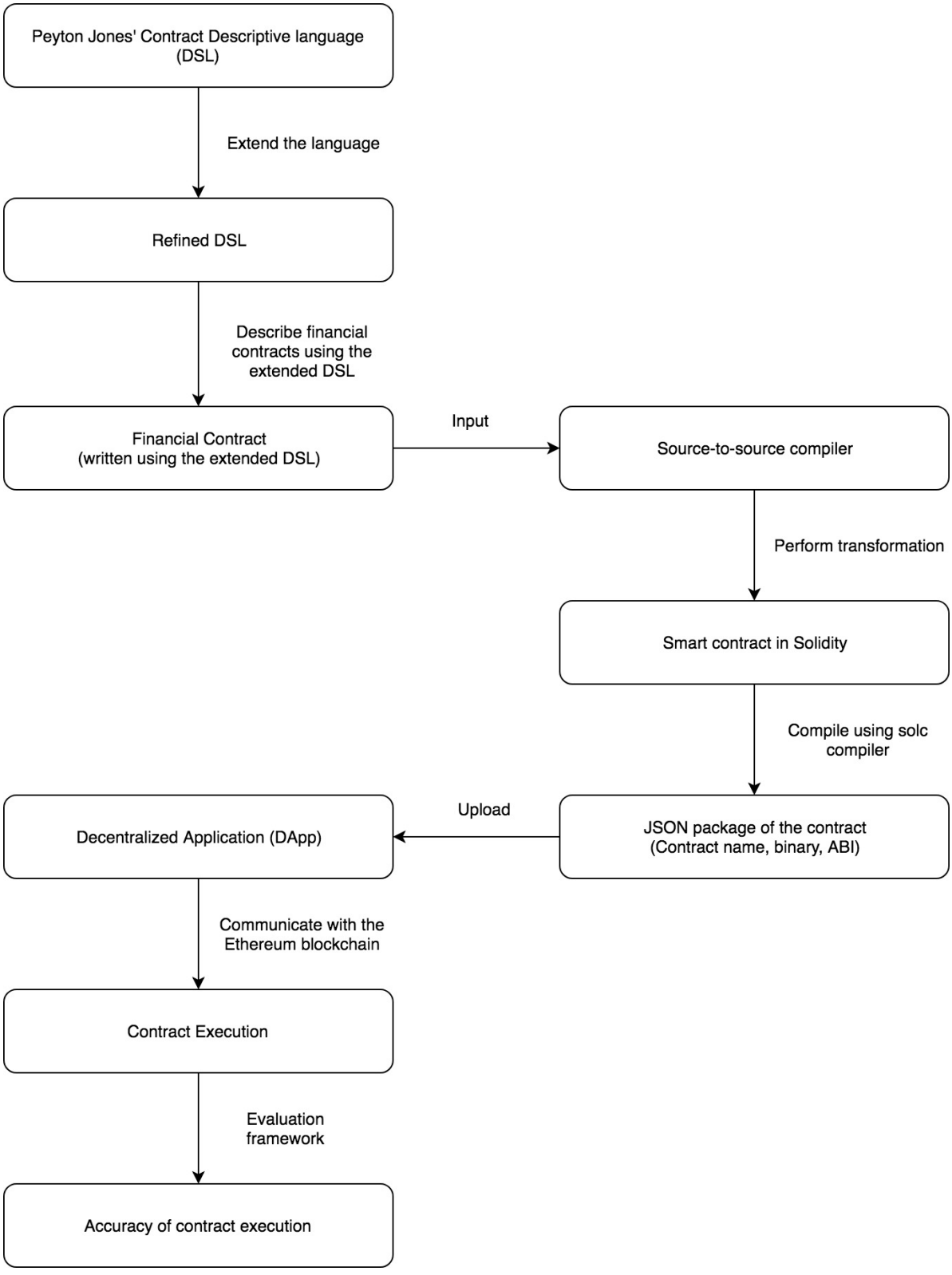


Figure 1.2: High-level diagram of proposed research methodology

1.6 Outline of the Dissertation

The dissertation is structured as follows. Chapter two explores the existing approaches to create DSLs for financial contracts and the re-implementations of financial contracts for the Ethereum blockchain. Chapter three describes the proposed research design and methodology. Potential ways of addressing the research problem is discussed in this chapter. Chapter four demonstrates the implementation details of the proposed methodology. Chapter five presents the evaluation model and the evaluation results of the proposed approaches. The last chapter, chapter six provides the conclusion of the dissertation and outlines the future work.

1.7 Delimitations of Scope

The proposed methodology of this study utilizes only the Peyton Jones' Contract descriptive language [2] as the source language for the compiler. The input to the compiler has been formatted in order to best suit the transformation. However, the formatting of the input does not change the semantics of the contract and as a result there is no particular effect to the processing of the contract. (This is described in detail in section 3)

The combinators proposed by Peyton Jones' et al. [2], the notion of horizon for contracts is utilized in order to generate an identical smart contract to the financial contract. Further, additional properties which are required for the smart contract to execute on the blockchain is embedded during the transformation.

The intention of this research is not to develop a new library or domain specific language specialized on describing specific types of contracts. Further, due to time constraints, it is outside the scope of this research to deliver a complete, optimized or readily usable compiler. It is also not in the scope of this research to address existing platform limitations and risks of the Ethereum blockchain when executing smart contracts.

1.8 SUMMARY

Financial contracts are widely used in the modern economy. The description of a financial contract includes a large vocabulary of financial jargon. Therefore, natural language is not suited to represent financial contracts due to its ambiguous nature. In order to address this problem Peyton Jones' et al. [2] introduced a domain specific embedded language (DSEL) which could describe financial contracts. Many other researchers who were inspired by this, continued to build DSELs which extended the language introduced by Peyton Jones et al. However, all these representations suffered from the same fundamental problem of the need to trust a third party to provide correct results of contract execution. Therefore, it is more desirable for a financial contract to facilitate autonomous, trustless execution in order to eliminate the risks involved in a centralized execution system.

This chapter mainly highlights the background of the research, the research problem and the research questions. Then the research was justified, the methodology was briefly described, the dissertation was outlined, and the limitations were given. On these foundations, the dissertation can proceed with a detailed description of the research.

Chapter 2 – LITERATURE REVIEW

2.1 INTRODUCTION

This chapter illustrates the current status of the research domain, especially targeting the *Peyton Jones' Contract Descriptive language* and how contract management is done. Next, the introduction of Ethereum, it's functionalities, smart contract applications in the past are thoroughly explored in section 2.2. Finally, an overview of source-to-source compiling done in previous researches are analyzed in section 2.3.

2.2 DSLS FOR FINANCIAL CONTRACTS

The past decade has been rich in terms of newly emerging DSLs for various purposes. Among them, several DSLs have emerged for financial contracts as well. Features and functionalities of few such DSLs will be explored in here.

The basic foundation to develop a DSL for financial contracts were laid by Peyton Jones et al. [2]. They introduced a combinator library to represent contracts and observables. The purpose of this library was to represent financial contracts in an unambiguous way. They also sketched an implementation of valuation semantics, using as an example a simple interest rate model and its associated lattice. A key feature of this library was that the combinators facilitated composability. Due to the compositional nature of the approach, complex contracts could be easily composed through the use of primitives for observables and primitives for contracts.

The combinator library was built using Haskell as the host language and as a result exploited Haskell's functional nature and lazy evaluation to a large extent.

An example contract in the Peyton Jones Contract Descriptive language takes the below format.

Zero Coupon Bond – `get ( truncate "date" ( scale k ( one USD )))`

Figure 1.1 in chapter 1 showed the combinators introduced by Peyton Jones et al. to describe contracts. Figure 2.1 shows the combinators introduced for observables.

`konst :: a -> Obs a`
`(konst x)` is an observable that has value `x` at any time.

`lift :: (a -> b) -> Obs a -> Obs b`
`(lift f o)` is the observable whose value is the result of applying `f` to the value of the observable `o`.

`lift2 :: (a->b->c) -> Obs a -> Obs b -> Obs c`
`(lift2 f o1 o2)` is the observable whose value is the result of applying `f` to the values of the observables `o1` and `o2`.

`instance Num a => Num (Obs a)`
All numeric operations lift to the `Obs` type. The implementation is simple, using `lift` and `lift2`.

`time :: Date -> Obs Days`
The value of the observable `(time t)` at time `s` is the number of days between `s` and `t`, positive if `s` is later than `t`.

Figure 2.1 – Combinators for observables

This paper also introduced the notion of a horizon for a contract. The horizon of a contract was defined as the latest date the contract can be acquired. After the horizon of a contract, it will be automatically void if not acquired by any party. Further improvements on the combinators were done through a book chapter [17] [18] released by Peyton Jones et al. at a later date where the notion of the horizon was discarded. However, our research has been based solely on the developments of the original composing contracts paper [2].

Based on the work of Peyton Jones et al. [2], DSLs were built as language extensions. One such approach was the DSL created by Gaillourdet [10]. The formal language developed in here was capable of expressing financial contracts themselves. She also introduced some denotational semantics which enables a notion of equivalence among contracts. A much broader DSL was developed by Andersen, et al. [19] which allowed description of multi-

party contracts as well. Additionally, they give denotational and operational semantics for that language, which allows to decide whether a trace of steps in the real world conforms to a specified contract or not. Hence, the real world applicability of the developed DSL has also been explored by them.

The central counterparty (CCP) of a financial contract is responsible for executing a financial contract. A central counterparty can make or break a contract as correct execution of contract logic is the sole responsibility of a central counterparty. An approach in the recent past has attempted to build a DSL to specify operations of a central counterparty [20] as well. This was conducted after identifying a research gap of CCP rules not been explored in the context of DSLs, even though CCPs were considered crucial for contract execution. In this study, it was proved that the Haskell combinator library introduced by Peyton Jones et al. [2] could be used to define CCP rules as well.

A financial contract has a dynamic timeline due to its uncertain nature. The parties involved in the contract needs to keep track of when the contract logic should be executed. If not, execution at the correct date may not happen due to misconducts of the CCP. Balalla V. R. et al. [21] conducted a study in the recent past to extract the contract calendar from a financial contract, based on the work of Peyton Jones et al. This study introduced a model with the capability of generating a calendar for a given contract. The model consists of a calendar definition, set of combinators for calendars and a set of evaluation semantics for the conversion from a contract to the calendar.

In an overall perspective, it could be seen that many researchers are interested in exploring the area of financial markets and financial contracts. Most of the researches in this area has been based on the influential paper by Peyton Jones et al. in developing precise combinators to represent contracts. Even though this study covered a major research gap at that time, it did not provide a solution to most risks involved in financial contracts. However, the declarative representation of contract proposed by Peyton Jones et al. opened many doors for further improvements on financial contracts.

2.2 Smart Contracts and its Applications

The idea of decentralization has been emerging in the world over the past decade. People are more keen on moving towards decentralization, leaving behind centralized authorities and autonomous power of a single entity. The concept emerged at a larger scale with the introduction of Bitcoin [4] in the year 2009. This new frontier of decentralization was adopted by many industries including finance. The next revolution in decentralization was the introduction of Ethereum [5]. The specialty of Ethereum beyond Bitcoin technologies was its scripting language Solidity [5] [22]. With the presence of Solidity it was possible to create pieces of codes that could run on the blockchain. These were called smart contracts.

A simple smart contract written in Solidity would have the following format.

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

A contract in the sense of Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain. The functions `set` and `get` can be used to modify or retrieve the value of the variable `storedDate`. This contract allows anyone to store a single number that is accessible by anyone in the world [8].

The Ethereum white paper [5], introduced many novel concepts including autonomous contracts, execution in a trustless environment, state transitions on the blockchain though message passing which facilitated users to build an entirely decentralized financial system.

Many researches were inspired by the concept of smart contracts after the introduction of Ethereum. Eksandari S. et al. [23] explored the feasibility of decentralized derivative

markets where they presented *Velocity*, a decentralized market deployed on Ethereum for trading a custom type of derivative option. Further, they have explored the security of smart contracts and the use of smart contracts in modelling financial systems. Through their study they have identified that smart contracts are a fascinating idea that can revolutionize the technology be removing the middleman involved in financial transactions. However, they have also stated that the infrastructure to build this system is still in a proof-of-concept state rather than in a production state in the present context.

Egelund-Müller et al. have also explored automated execution of financial contracts on the blockchain [24]. The paper investigates financial contract management on distributed ledgers and provides a working solution implemented on the Ethereum blockchain. The system is based on a domain-specific language for financial contracts that is capable of expressing complex multi-party derivatives and is conducive to automated execution. The authors also propose an architecture for separating contractual terms from contract execution. However, they have completely ignored the use cases of a DSEL to represent a financial contract and have looked at a re-implementation of financial contracts on the Ethereum blockchain. This have eliminated the composable nature of contracts in the original work of Peyton Jones et al.

Safer smart contracts through type-driven development has been explored by Pettersson et al. [14] in the past where they have shown how dependent and polymorphic types can make smart contract development safer. This has been demonstrated by using the functional language Idris to describe smart contracts on the Ethereum platform. Same as the previous approach, they have too implemented a novel functional language for smart contract development on Ethereum. They have not explored the possibilities of using existing DSELs for the task.

As such, the smart contract domain has been thoroughly explored by researchers as well, similar to the financial contract domain. However, the approaches in all previous work has been independent to the respective domain. Bridging the gap of these two domains haven't been proposed by any previous work in the domains.

2.3 Source-To-Source Compiling

The objective of this study is achieved by building a source-to-source compiler (also known as a transpiler). The frontend of the compiler was built using ANTLR4. This is a popular, commercial tool which has been used in many previous work when conducting language translations. One such study by Cheng Zhou [25] was the implementation of a source to source compiler that translates Fork language to REPLICA baseline language. The Fork language is a high-level programming language designed for the PRAM (Parallel Random Access Machine) model. The baseline language is a low-level parallel programming language for the REPLICA architecture which implements the PRAM computing model. To support the Fork language on REPLICA, a compiler that translates Fork to baseline was built in this study.

The compilation process of the Fork to baseline compiler is described in figure 2.2.

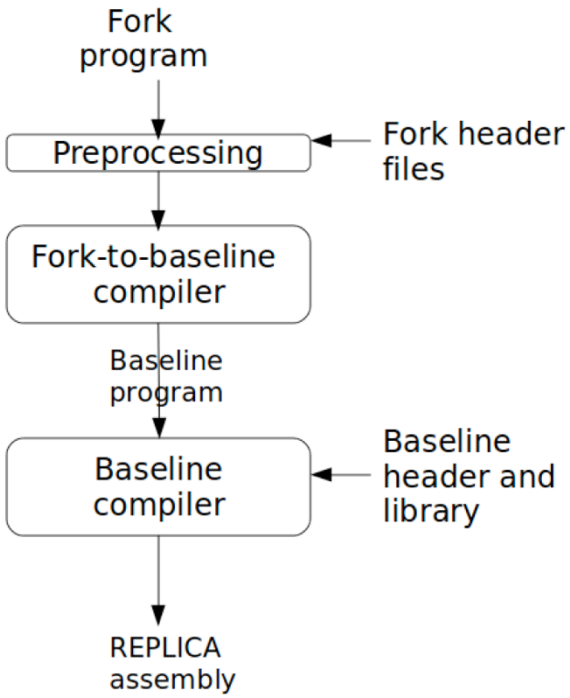


Figure 2.2 - Compilation process of the Fork to baseline compiler

ANTLR was used in this study to generate the parse tree and the intermediate representation (IR). The following figure 2.3 shows the translated baseline IR.

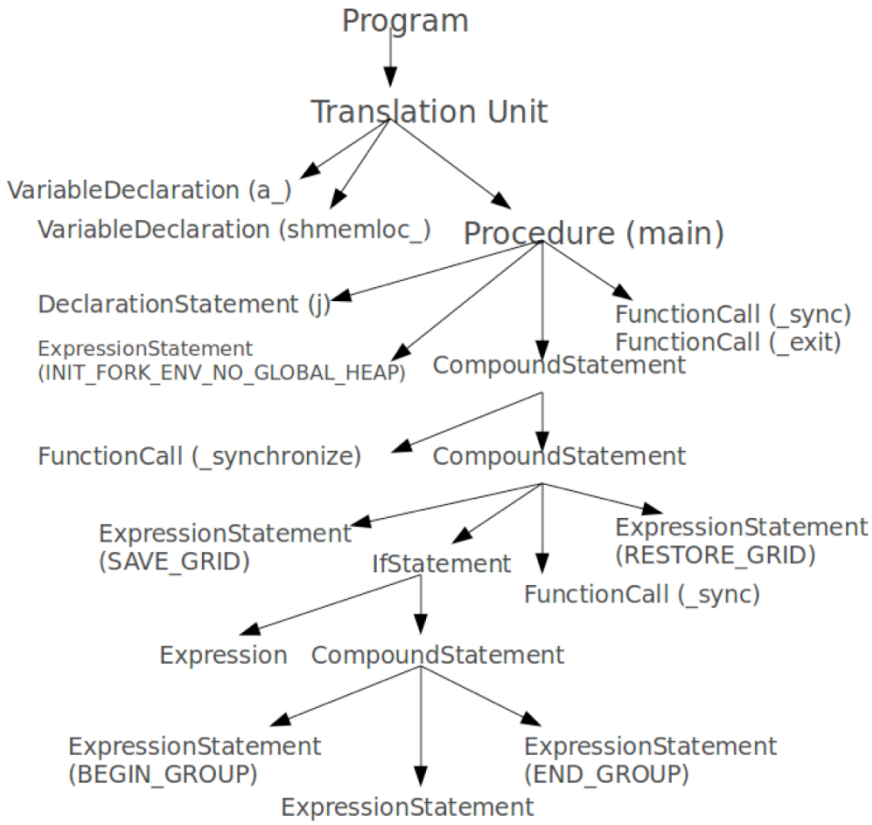


Figure 2.3 – Translated baseline IR

They have concluded that the Fork language is supported for the REPLICA architecture by building the source to source compiler and supporting libraries. This study showcases the significant use of ANTLR [26] for source-to-source compiler building.

Chapter 3 – DESIGN

3.1 INTRODUCTION

This chapter mainly elaborates the proposed solutions to the research problem. It consists of four major sections, namely; Research Design, Extending the Peyton Jones' language, Compiler Design and Smart Contract Design.

3.2 RESEARCH DESIGN

The research design comprises of three main sections: Extending the DSL, Source-to-source compilation and the Transformed contract deployment. The final implementation of the compiler and the Ethereum client is called *Synergy*. Figure 3.1 showcases a high-level diagram of the relationship of these sections.

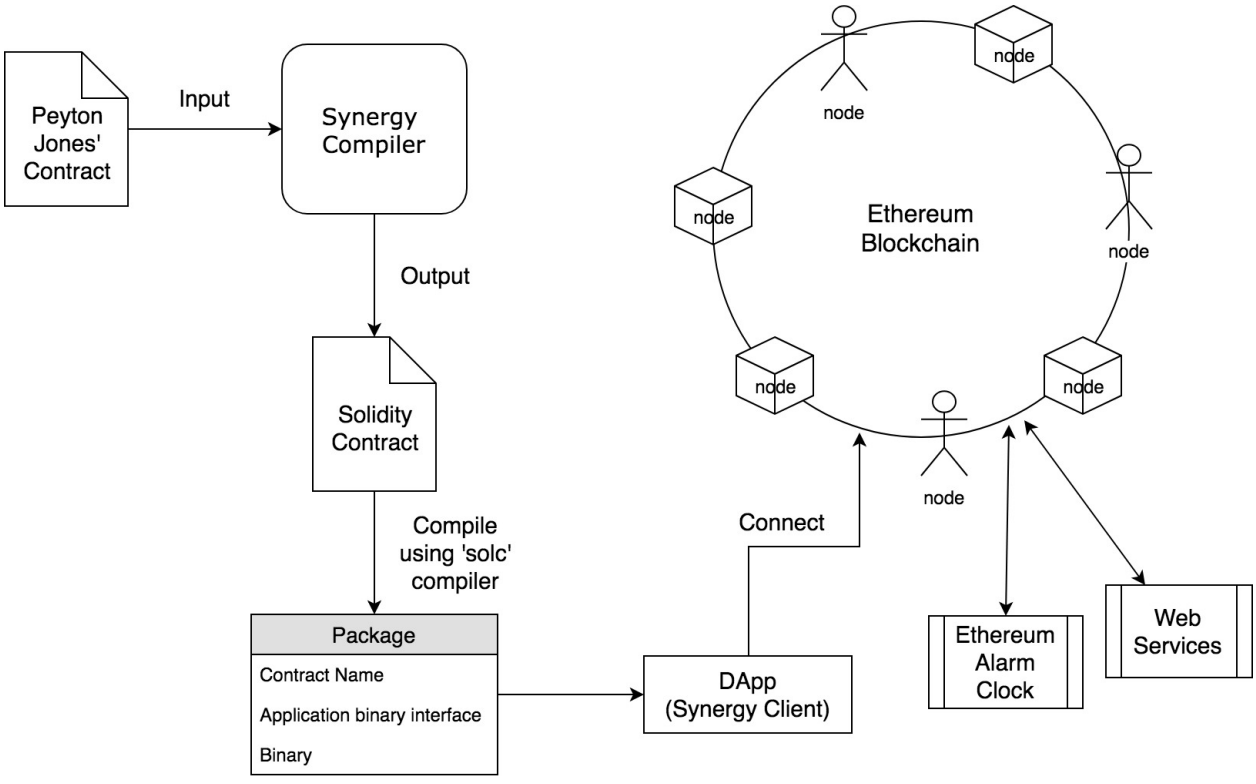


Figure 3.1: High-level architecture of the proposed solution

The first phase of the research focused on building a source-to-source compiler in order to transform a contract written in Peyton Jones' contract descriptive language to Solidity.

3.3 Extending the Peyton Jones' CDL

The current implementation of the Peyton Jones' CDL is embedded in the functional programming language, Haskell. In order to write a compiler, the combinators introduced by Peyton Jones et al. [2] needed to be embedded in Haskell in an extensible manner.

There are two key methods of implementing a DSL: shallow embedding and deep embedding [27]. These were introduced as two approaches to embedding a hardware description language (HDL) in higher-order logic (HOL). The features of shallow and deep embedding are shown in Table 3.1 below.

Table 3.1: Features of shallow and deep embedding of a DSL

Shallow Embedding	**Deep Embedding**
Represents elements by their semantics (A one-to-one mapping from elements to semantics)	Represents elements by how they are constructed
DSL constructs are composed purely of host language constructs	Program exists as data (E.g.: AST)
Uses the interpreter of the host language	Host language implements an interpreter
More elegant (when it works out)	Easier to extend (To add new operations, run functions, optimizations, etc.)

When considering Haskell, it is understood that a shallow embedding implies the use of functions (or more commonly known as combinators) to represent operations in the DSL. The Peyton Jones' CDL is one of the best examples for such a shallow embedding. In order to build a compiler for this language, it was required to have different representations of the same function (i.e.: Contract Description, Contract Abstract Syntax Tree, etc.). However, different definitions of the same function led to complications because it had to maintain a huge tuple for all the definitions. As a solution to this, the original language proposed by

Peyton Jones et al. was extended to a deep embedding in Haskell. In here, the DSL was represented as a Haskell data structure, which could be interpreted by other functions. (Discussed in detail in section 4)

3.4 Compiler Design

3.4.1 Source Language

The choice of the source language for this study, the Peyton Jones' CDL was the foundation to formulate a process to automate financial contracts. This language represents financial contracts in an unambiguous manner and is composable in nature, which allows to create complex contracts as a combination of simple primitives defined as combinators.

3.4.2 Target Language

When designing a compiler for a source language to be executed in the EVM, there are four possible target languages for the code generator: directly into assembly instructions/bytecode, LLL (Low-Level LISP), Serpent and Solidity. Conversion to assembly instructions directly may impose some risks in using this compiler in future, as the EVM is subject to changes as it improves in the future. Therefore, conversions directly to the bytecode may become obsolete. A comparison of the available higher-level languages are shown in Table 3.2 below.

Table 3.2: Comparison of the languages for smart contract development

	Data Structures	**Contract Functions**	**Macros**	**Targets**
LLL	No	No	Yes	Bytecode
Serpent	Yes	Yes	Yes	LLL
Solidity	Yes	Yes	No	Bytecode

Both LLL and Serpent are lower-level languages than Solidity where LLL has the advantages of lower runtime overhead and smaller binary output than code written in the

other two languages. However, due to the low level nature of LLL the following concerns prevail.

- The only way to access persistent memory is through direct reading and writing to addresses.
- Complex data types/structures such as arrays and lists are not supported. Therefore, allocating and aligning memory positions for complex data types needs to be done explicitly, which is non-trivial.
- There is no concept of functions. Therefore, all programs have a single entry point, requiring all user-defined functions to be implemented as conditional checks [28].

Even though Serpent's compiler handles some of the above while still giving flexibility to access assembly instructions, it is not maintained and has an increase in runtime overhead and code size compared to Solidity. Considering all of the above, Solidity has been chosen as the target language for this study as it is the most-widely used, maintained and feature-rich language, with syntax similar to JavaScript.

3.4.3 Compiler Construction

The compiler of this study has been designed from the beginning to support new compiler backends. Even though Solidity has been chosen as the target language, it is possible to develop a new compiler backend and use the intermediate representation generated by the compiler frontend to convert the financial contract to other suitable languages as well. The main phases of the compiler are as follows.

Tokenization and Parsing – A grammar was written for the Peyton Jones' CDL. Based on the grammar, the lexer and the parser was generated. The contract input (composed of primitive combinators) was given to the lexer, which tokenized the input string into tokens. Next, the tokens are given as an input to the parser, which builds the parse tree for the given input.

Intermediary transformation – The nodes of the Abstract Syntax Tree (AST) is obtained from the parse tree in order to facilitate the target language code generation.

Tree walker – The abstract syntax tree is walked using a tree walker which conducts the transformations of the contract combinators, mapping the necessary logic of the Haskell combinators to the target language. The tree walker performs the final code generation of the Solidity source code for a contract.

The complete compiler design is shown in Figure 3.2 below.

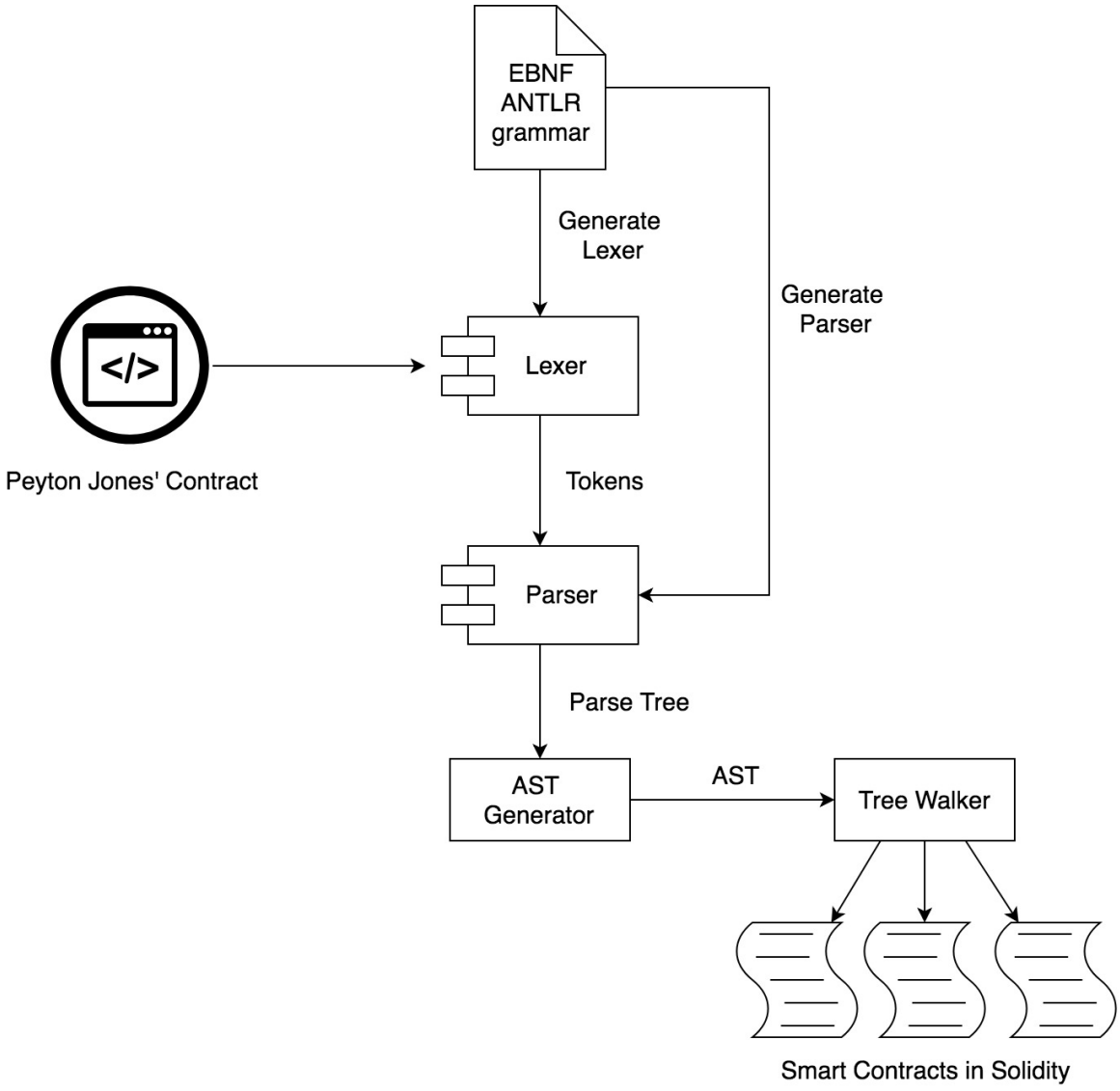


Figure 3.2: The design of the compiler which transforms Peyton Jones' contracts to Solidity

3.5 SMART CONTRACT DESIGN

The design and the specificities of the transformed smart contract were decided based on the nature of the parse tree generated. There are many approaches that could be followed in order to generate the Solidity source code. One such approach is to have a series of nested Solidity contract objects for each combinator in the contract input. However, this method would drastically increase the transaction cost when executing the contracts in the Ethereum blockchain, as each contract creation would cost a considerable amount of gas. Therefore, the proposed methodology in this study minimizes the number of nested contracts created when transforming a contract to Solidity.

The nature of a complex specific contract is shown in Figure 3.3 below.

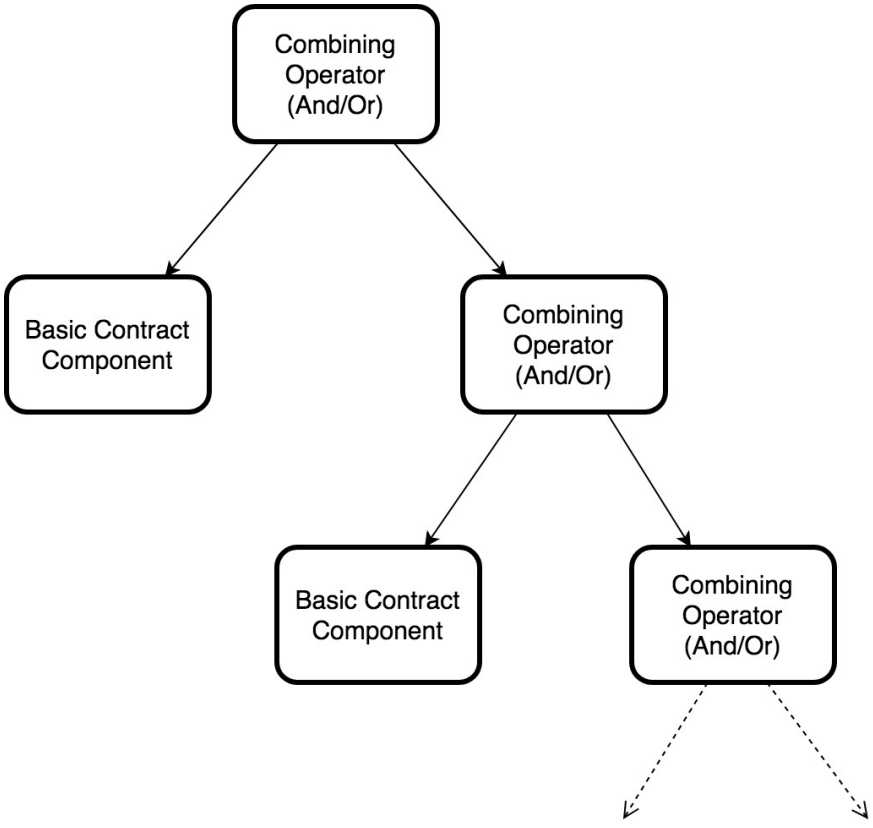


Figure 3.3: The structure of a complex specific contract

A basic contract component would have a set of standard basic primitives (combinators). The following figure 3.4 showcases the structure of a basic contract component (extracted from the AST). Depending on the input, a basic contract component would consist of one or more combinators stated in the structure below.

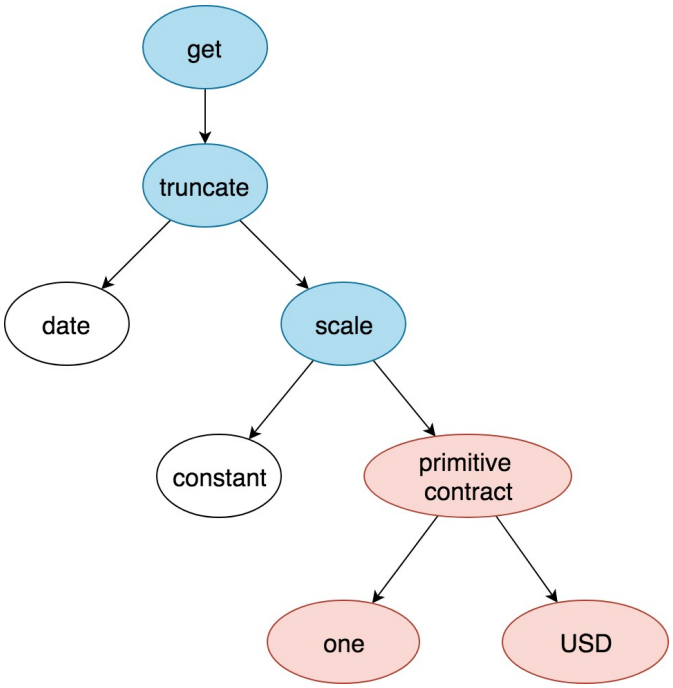


Figure 3.4: Structure of a basic contract component

When a 'get' combinator is obtained by the tree walker while traversing the AST, a Solidity source file is generated for the basic contract component. The required logic of the basic contract will be embedded in the Solidity source. Apart from the contract logic expressed in the Peyton Jones' contract input, there are additional information required by the Ethereum blockchain in order to execute a contract on the blockchain. In order to embed the additional information required and also to modularize the contracts, two external contracts were introduced: Synergy Marketplace Contract and the Base Contract.

The base contract provides a skeleton to the specific contracts are generated based on the Peyton Jones' contract input while the Synergy Marketplace contract achieves contract modularization (Implementation explained in detail in section 4). Modularized contracts are

of vital importance to reduce execution cost of a contract on the blockchain. The additional information required by the smart contract and how they are facilitated are shown in Table 3.3 below.

Table 3.3: Additional information required to execute smart contracts

Information	***Description***	***Facilitation***
Contract parties	In order to execute a contract on the blockchain, the account addresses of the parties involved in the agreement is required	**Contract owner**: Embed in the Synergy marketplace contract at the time of deployment. **Contract Holder**: Embed in the Synergy marketplace contract at the time of proposing the contract to the holder
Functions to propose and sign contracts, receive and send currency, etc.	Contract enforcement happens via function calls on the Ethereum blockchain	The Synergy marketplace contract would have implementations for all the required functions.
Self-destruct or kill a contract once the horizon of a contract is exceeded	After the horizon of a contract, it should be made void automatically as it should not be possible to acquire a contract after its horizon	The base contract includes a function for destroying the contract. kill() function
Contract storage to store contract addresses and balances	This is required in order to keep track of all the contracts and user balances	Member variables are introduced in the Synergy marketplace contract
Execute contracts at a future date	Smart contracts need to be executed automatically without the involvement of a third party. Therefore, the future execution dates need to be monitored to execute contract logic at the correct date.	The Ethereum Alarm Clock[1] A service that allows to schedule transactions to be executed at a future date on the Ethereum blockchain. Up-front payments are done for gas costs of transactions.

[1] **Ethereum Alarm Clock** https://www.ethereum-alarm-clock.com/

Figure 3.5 and Figure 3.6 shows how the specific contract is created in Solidity and the format of the transformed Solidity contract respectively.

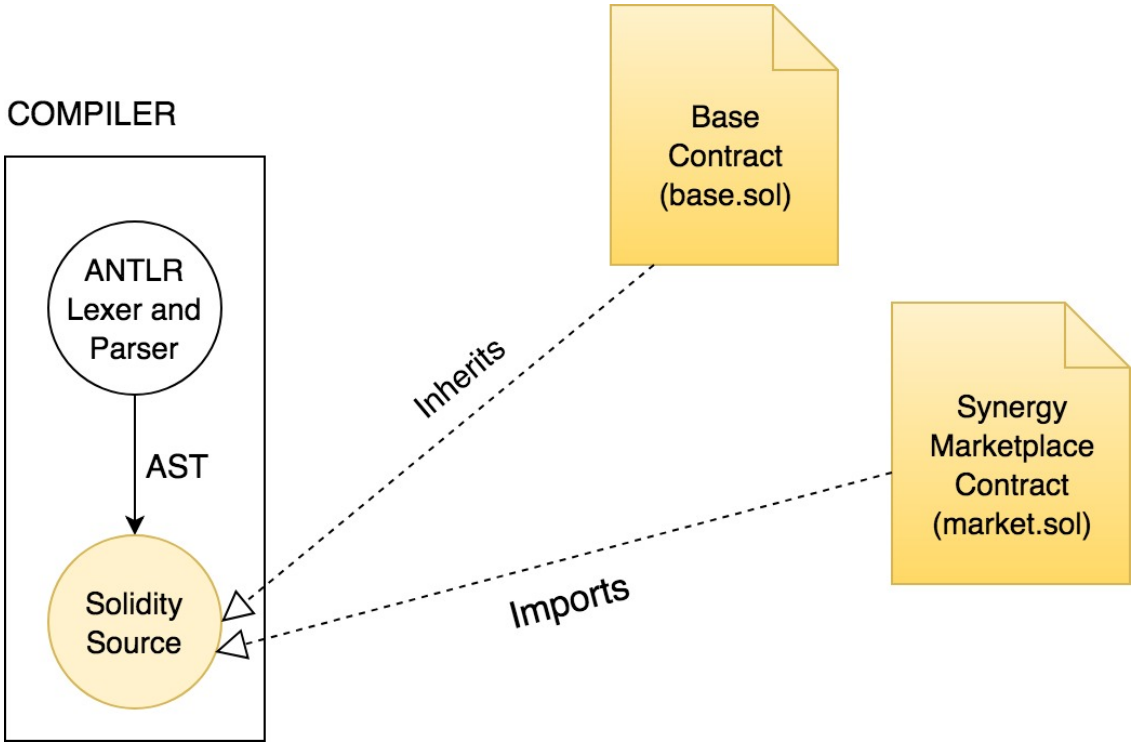


Figure 3.5: Solidity source generation

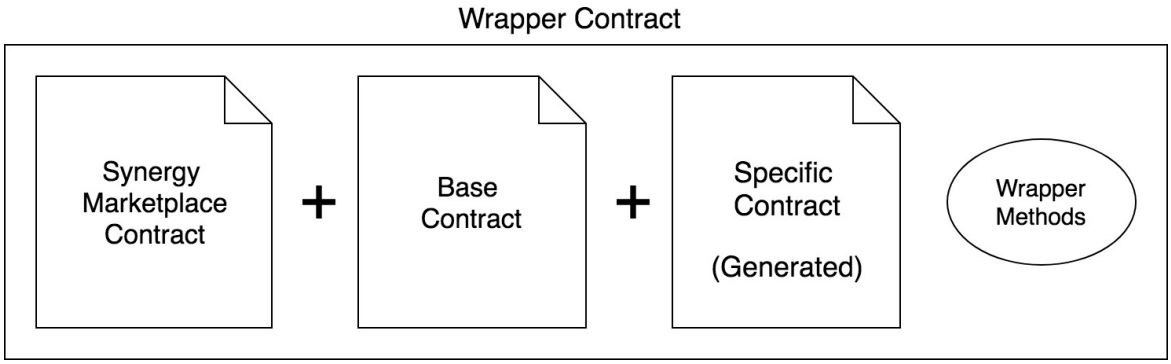


Figure 3.6: Final Solidity Contract Code

After the Solidity source file is generated, it is uploaded and deployed to the blockchain using the decentralized application (DApp) which was built. The contract code lives in the blockchain until its logic is executed with the assistance of the Ethereum Alarm Clock service. More details about the contract execution is discussed in section 4.

3.6 JUSTIFICATION FOR THE METHODOLOGY

The methodology proposed in this study is built based on a DSEL, the Peyton Jones' contract descriptive language. The main reason to choose a DSEL opposed to a standalone language is that it allows to reuse the host language's advanced type system and flexible syntax. Further, it has been identified that a financial contract is best represented using a domain specific language. This is due to a characteristic property of DSLs where it is possible to write code in terms closer to the level of abstraction of the initial problem domain. However, since a DSL has a high level of abstraction related to the domain, interoperability is a crucial problem. Therefore, DSLs are often integrated to a host language (Haskell in here) and are converted to DSELs.

Peyton Jones' contract descriptive language has been the basis of many work in this particular area [10] [15].

A deep embedding was done to the original language proposed by Peyton Jones' et al. because, representing the language in a different domain required different representations of the same function.

Previous researchers have implemented methodologies in re-writing financial contracts as smart contracts in order to facilitate autonomous execution. Opposed to such approaches, this study mainly proposed a transformation of traditional financial contracts written in the Peyton Jones' CDL to smart contracts written in Solidity due to few key reasons.

- The need of preserving the composable nature of the DSEL
- Make use of Haskell's advanced type system and flexibility
- Financial contracts are best represented using a DSEL
- Rewriting existing contracts would incur additional overhead

3.7 SUMMARY

This chapter provided a detailed description on the research design. It encompasses three main phases; namely Extending the Peyton Jones' CDL, Compiler Design and Smart Contract Design. The language is extended in order to support multiple definitions of the same functions. Once the compiler design was done, the smart contract was generated with various embeddings in order to suit the Ethereum blockchain. The generated contract is then deployed using a decentralized application (DApp) to the blockchain.

Chapter 4 - IMPLEMENTATION

4.1 INTRODUCTION

This chapter provides implementation details of the proposed solutions. Section 4.2 describes the software tools utilized for the implementation process, section 4.3 illustrates how the Peyton Jones' language was extended, section 4.4 describes the implementation details of the compiler and section 4.5 explains how the transformed contracts are deployed and executed on the blockchain.

4.2 TECHNOLOGIES AND SOFTWARE TOOLS

The Peyton Jones' CDL extension was done using the *Haskell* language (GHC 8.4.3, Stack 1.7.1). The frontend of the compiler was built using Another Tool for Language Recognition V4 (ANTLR4). The BNF grammar for the Peyton Jones' CDL was written based on ANTLR. The lexer and the parser was generated in JavaScript. The tests on contracts were run using TestRig[2]. ANTLR provides a single consistent notation for specifying lexers, parsers, tree parsers, etc. opposed to other tools available. The TestRig component of ANTLR allows to obtain a graphical representation of the parse tree as well. A mixture of go-ethereum (geth), Parity, Mist, Remix and MetaMask were used to simulate transformed contracts and interact with an Ethereum blockchain (Ropsten Testnet).

4.3 EXTENDING THE LANGUAGE

4.3.1 SHALLOW EMBEDDING

The Peyton Jones' CDL has utilized a shallow embedding to include the combinators in the host language Haskell. However, this method of embedding is suited only if one representation of a particular contract is required.

```
type Contract = String
```

[2] TestRig https://www.antlr.org/api/JavaTool/org/antlr/v4/gui/TestRig.html

```
and :: Contract -> Contract -> Contract
and c1 c2 = "And(" ++ c1 ++ "," ++ c2 ++ ")"
```

In here, the type keyword is used to cast the type 'Contract' to type 'String'. This representation prints the complex contract given, as a string. If another representation is required, for example, to count the number of combinators a particular contract has, the logic should be re-implemented.

```
type Contract = Int

and :: Contract -> Contract -> Contract
and c1 c2 = 1 + c1 + c2
```

This brings out a conflict as the same function cannot have two different representations. Therefore, if shallow embedding is used in order to have different representations of the same function, all the interpretations needs to be returned within the function.

```
type Contract = (String, Int)

and :: Contract -> Contract -> Contract
and c1 c2 = ("And(" ++ c1 ++ "," ++ c2 ++ ")", 1 + c1 + c2)
```

This method becomes very cumbersome if there are a large number of interpretations for the same function. Therefore, this is not suitable because having to maintain a huge tuple for every possible permutation of options is not a practical solution.

4.3.1 Deep Embedding

In order to transform a contract written in Peyton Jones' CDL, it is required to have different interpretations for the same contract. These representations could be the contract text as Haskell combinators, contract parse tree, contract abstract syntax tree for different types of contracts. In order to obtain all these interpretations for the same contract, it is clear that a deep embedding is required because in a deep embedding the DSL is represented as a

Haskell data structure. This data structure can be interpreted by other functions making it possible to have several interpretations on the same function.

This particular data declaration mentions that a value of type '`Contract`' can be constructed by calling `Zero`, `One USD`, `Give c`, etc.

```
data Contract contract
    = Zero
    | One  Currency
    | Give contract
    | Get contract
    | And  contract contract
    | Or   contract contract
    | Scale   (Obs Int) contract
    deriving (Eq, Show)
```

The deep embedding done above solves the problem of multiple interpretations on the same function. The `render` and `count` interpretations were modified as follows.

```
render :: Contract -> String

render Zero = "Zero"
render (One currency) = "One(" ++ show currency ++ ")"
render (Give contract) = "Give(" ++ render contract ++ ")"
render (Get contract) = "Get(" ++ render contract ++ ")"
render (And contract1 contract2)="And("++render contract1++","++render contract2++")"
render (Or contract1 contract2) = "Or("++render contract1++","++render contract2++")"
```

```
count :: Contract -> Int
count Zero = 1
count (One _) = 1
count (Give contract) = 1 + (count contract)
count (Get contract) = 1 + (count contract
count (And contract1 contract2) = 1 + (count contract1) + (count contract2)
count (Or contract1 contract2) = 1 + (count contract1) + (count contract2)
```

However, due to the deep embedding the appearance of the original contract may change. Therefore, constructors were defined in order to retain the original syntax.

```
zcb :: Date -> Double -> Currency -> Contract
zcb t x k = scaleK x (get (truncate t (one k)))

zcb' :: Date -> Double -> Currency -> Contract
zcb' t x k = Scale (Const x) (Get (Truncate t (One k)))
scaleK x c = Scale (Const x) c
get c = Get c
truncate t c = Truncate t c
one k = One k
```

4.4 Source-to-Source Compiling

Once the Peyton Jones' language was extended to facilitate multiple interpretations, the transformation of the contract was done in the following steps.

- EBNF grammar using ANTLR4 for the Peyton Jones' language
- Lexer generation
- Parser generation
- Built the abstract syntax tree generator
- Built the tree walker
- Generation of the Solidity version of the financial contract
    - Creation of the base contract
    - Creation of the Synergy marketplace contract
    - Generation of the specific contract with a wrapper

4.4.1 ANTLR Grammar for the Peyton Jones' Language

The first phase of building a source-to-source compiler (also known as a transpiler) is developing the grammar for the source language; Peyton Jones' CDL in this particular

study. The grammar written for the language consists of two components; the lexer rules and the parser rules. The lexer rules are responsible of generating the lexer which is used for tokenization of the language input and the parser rules are responsible for generating the parser which is used to parse the tokens from the lexer and build the parse tree for the input.

4.4.2 Lexer and Parser Generation

The lexer rules for the Peyton Jones' contract descriptive language is as follows.

```
Operator : '`and`' | '`or`';

Currency : 'USD' | 'GBP' | 'ETH';

OneKeyword : 'one';

ZeroKeyword : 'zero';

//Keywords
Scale : 'scale';
Give : 'give';
And : 'and';
Or : 'or';
Truncate : 'truncate';
Then : 'then';
Get : 'get';
Anytime : 'anytime';

Date : DateInString;
DateInString : StringLiteral;

ObsDouble //Decimal Number
  : [0-9]+ ( '.' [0-9]* )? ( [eE] [0-9]+ )?;

//Contract variables
ID : IdentifierStart IdentifierPart* ;
fragment
IdentifierStart : [a-zA-Z] ;
fragment
IdentifierPart : [0-9] ;
StringLiteral : '"' DoubleQuotedStringCharacter* '"'
```

```
              | '\'' SingleQuotedStringCharacter* '\'';
fragment
DoubleQuotedStringCharacter : ~["\r\n\\] | ('\\' .);
fragment
SingleQuotedStringCharacter : ~['\r\n\\] | ('\\' .);
BACKQUOTE : '`';
WS : [\t\u000B\u000C\u0020\u00A0]+ -> channel(HIDDEN);
```

The parser rules for the Peyton Jones' contract descriptive language is as follows.

```
complexContract
    : basicContract | basicContract Operator complexContract;

basicContract
    : basicPrimitive | compositePrimitve;

basicPrimitive
    : zeroContract #ZERO | oneContract #ONE;

zeroContract
    : ZeroKeyword;

oneContract
    : OneKeyword Currency;

compositePrimitve
    : scale #SCALE_CONTRACT
    | give #GIVE_CONTRACT
    | truncate #TRUNCATE_CONTRACT
    | then #THEN_CONTRACT
    | get #GET_CONTRACT
    | anytime #ANYTIME_CONTRACT;

scale
    : Scale ObsDouble '(' complexContract ')';

give
    : Give '(' complexContract ')';

truncate
    : Truncate Date '(' complexContract ')';
```

```
then
    : Then '(' complexContract complexContract ')';

get
    : Get '(' complexContract ')';

anytime
    : Anytime '(' complexContract ')';
```

The parser generated by the above parser rules is a LL(1) parser. It reads the input from left-to-right, descend into parse tree children from left-to-right and utilizes a single lookahead token. A parser with a single lookahead token is one of the weakest forms of parsers. However, it is sufficient for this purpose as the Peyton Jones' CDL has a handful of combinators.

ANTLR4 does not support left recursion[3]. Therefore, the input to the compiler should be presented with prefix operations. An example input to the compiler is as follows.

European Option

```
get ( truncate "15 Jan 2019" ( or ( scale 100 ( get ( truncate "30 Jan 2019" ( one GBP ))) zero )))
```

The entry point to the compiler is the `complexContract` parser rule. The parser generated from the above parser rules is compatible with basic contracts as well as complex contracts. Figure 4.1 and 4.2 shows the railroad diagrams for the parser rules which accepts complex contracts and basic contracts respectively.

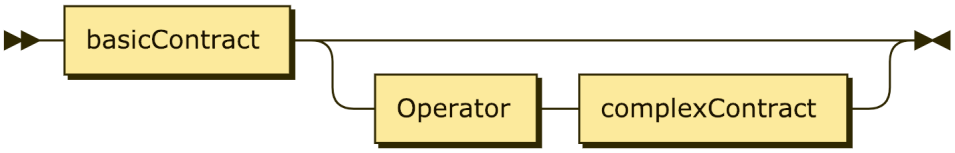


Figure 4.1: Complex contract railroad diagram

[3] **Left Recursive rule:** A rule that invokes itself without consuming a token.

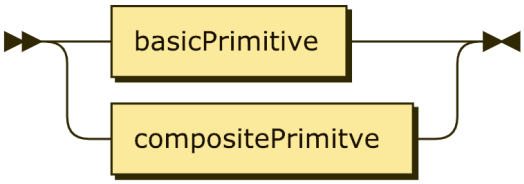


Figure 4.2: Basic contract railroad diagram

To make parsing decisions, the parser tests the current lookahead token against the alternatives' lookahead sets. The lookahead set computation (FIRST and FOLLOW) is done by ANTLR4. Naturally, the lookahead sets predicting the alternatives are disjoint in here as what has been built is a deterministic LL(1) parser.

4.4.3 Abstract Syntax Tree and Tree Walker

The next step of the compiler implementation was to build the intermediate representation (Abstract Syntax Tree - AST) for the language. The AST holds key tokens and records the grammatical relationships. An AST was built because it has the following advantages over a parse tree.

- Easier and faster to walk
- Easier to identify subtrees
- Dense – No unnecessary nodes
- Emphasizes operators, operands and the relationship between them rather than the artifacts of the grammar. Therefore, it is insensitive to changes in the grammar. (Easily maintainable if the Peyton Jones' CDL improves in the future)

The following code segment represents the parse tree traversal in order to build the AST.

```
function recurse (tree, list) {
    let a = ['(', ')', '<EOF>'];
    if(tree.getChildCount() == 0) {
        if(!a.includes(tree.getText())) {
            list.push(tree.getText());
            return;
        }
    }
    for(let i=0; i < tree.getChildCount(); i++) {
        recurse(tree.getChild(i), list);
    }
}
```

Once the AST was generated, a tree walker was written in order to traverse the AST and generate the Solidity code based on the tree structure of the input. It was built to collect the necessary information of all the combinators of the input. The contract Solidity source was created when the tree walker meets a `get/give` combinator. The Solidity source of the contract includes the instantiated base contract, imported Synergy marketplace contract, implementation of the specific contract and the wrapper methods in a single file.

The horizons of contracts are monitored at `get/give` combinators if the contract is a basic contract while it is monitored at `and/or` operators if the contract is a complex contract (Discussed in detail in the next sub-section). Ultimately, the information about contract(s) at the root of the tree is returned to a wrapper contract, which provides the entry point for contract execution. The wrapper is needed for contract enforcement when it is deployed to the blockchain. (Code listings included in Appendix C)

4.4.4 Base Contract

The specific Solidity source is created by inheriting the base contract referenced above. It is an abstract base class for the transformed contracts that allows to interact with a contract in a standardized manner. The base contract has few important features. It exposes a proceed() method. The actual behavior of this function is implemented when the specific

contracts inherit the base contract. Further, it provides a modifier to check whether a particular contract is alive before any function is applied. This will only permit a function to execute if the `_alive` member variable is true.

```
contract BaseContract {

    enum KillReason {EXECUTED, UNTIL, HORIZON, FAILED}
    event Killed(BaseContract.KillReason killReason);
    Marketplace public marketplace_;
    int public scale_;
    address public creator_;
    bool public alive_ = true;
    constructor(Marketplace marketplace, int scale) public {
       marketplace_ = marketplace;
       scale_ = scale;
       creator_ = msg.sender;
    }
    function proceed() public;

    function receive(Marketplace.Commodity commodity, int quantity) internal alive {
        marketplace_.receive(commodity, quantity);
    }

    function kill(BaseContract.KillReason killReason) internal alive {
        alive_ = false;
        emit Killed(killReason);
    }

    modifier alive {
        require(alive_);
        _;
    }
}
```

The function to kill the contract is protected by the internal keyword. When this keyword is present in a function, it could only be called by the smart contract itself. Therefore, one smart contract cannot kill another smart contract.

4.4.5 SYNERGY MARKETPLACE

As mentioned in section 4.4.3, a specific contract imports certain functionality from an external contract; Synergy Marketplace Contract. This particular contract plays a major role in modularizing the codebase of the compiler and contract execution. The functions to all contract executions are included in the marketplace contract. Further, information such as contract addresses, user balances, commodities, etc. are also kept in here. For every contract creator, an instance of the marketplace is deployed to the blockchain before deploying any specific contracts.

The marketplace contract has few important functions which assists contract execution. Initially, a couple of data structures are declared. `Commodity` represent the commodities that can be traded in this market and `ContractMetadata` has information about current state of a contract. The parties involved in a contract are stored in here, where `Counterparty` is the party that sells the contract while `Holder` is the party that buys the contract. Few member variables are also stored in here. The `contracts_` mapping keeps track of all the contracts that have been proposed through the marketplace, while the `balances_` mapping keeps track of each individual user's balance in each commodity.

```
pragma solidity ^0.4.23;
pragma experimental ABIEncoderV2;

contract Marketplace {
      enum Commodity {USD, GBP}

      struct ContractMetadata {
          address counterparty;
          address holder;
          address creator;
          bool signed;
      }

      event Proposed(address contractAddress, address indexed to);
      event Signed(address contractAddress);

      address public creator_;
```

```
    mapping(address => ContractMetadata) public contracts_;
    mapping(address => mapping(uint => int)) public balances_;

    constructor() public {
        balances_[msg.sender][uint(Commodity.USD)] = 0;
        balances_[msg.sender][uint(Commodity.GBP)] = 0;
        creator_ = msg.sender;
    }
```

Few key functions in the marketplace contract enables contract execution on the blockchain. The `propose` method allows a counterparty to propose (sell) a contract to another party. Once a contract is proposed, it creates a contract metadata object with the necessary information and emits an event. When an event is emitted the information gets stored in a transaction log. This can be checked by other functions or from outside the contract. The `sign` method then allows the holder (to whom the contract is proposed) to agree to the contract, which starts executing immediately. During the signing process of the contract the state of the contract is changed to `'signed'`.

```
function propose(address contractAddress, address to) public {
    require(contractAddress.creator_() == msg.sender);
    require(!contracts_[contractAddress].signed);
    contracts_[contractAddress] = ContractMetadata(msg.sender, to, msg.sender,
false);
    emit Proposed(contractAddress, to);
}

function sign(address contractAddress) public {
    require(msg.sender == contracts_[contractAddress].holder);
    require(!contracts_[contractAddress].signed);
    contracts_[contractAddress].signed = true;
    BaseContract baseContract = BaseContract(contractAddress);
    baseContract.proceed();
    emit Signed(contractAddress);
}
```

The following functions can only be called by signed contracts. The receive method allows the counterparty to transfer a certain amount of the commodity traded, to the holder. In the case of a nested contract, multiple instances of contracts would be created. The `get` method

is used to assign the authorities from the parent contract to the child contract whereas the `give` method does the same with the counterparty and holder reversed.

```
function receive(Commodity commodity, int quantity) public {
      ContractMetadata storage c = contracts_[msg.sender];
      require(c.signed == true);
      balances_[c.counterparty][uint(commodity)] -= quantity;
      balances_[c.holder][uint(commodity)] += quantity;

function delegate(address newContract) public {
      require(contracts_[msg.sender].signed == true);
      contracts_[newContract] = ContractMetadata(
          contracts_[msg.sender].counterparty,
          contracts_[msg.sender].holder,
          msg.sender,
          true
      );
      emit Delegated(msg.sender, newContract); }
function give(address newContract) public {
     require(contracts_[msg.sender].signed == true);
     contracts_[newContract] = ContractMetadata(
         contracts_[msg.sender].holder,
         contracts_[msg.sender].counterparty,
         msg.sender,
         true
     );
     emit Delegated(msg.sender, newContract);
}
```

4.4.6 Solidity Source Creation

The final step of the contract transformation is generating the Solidity code for the contract. A wrapper contract was introduced in order to wrap the base contract, marketplace contract and the specific contract together. The wrapper contract includes few wrapper methods which assists in contract invocation. The contract logic is adequately combined in here.

The most important function in the wrapper contract is the `proceed` method, which is the entry point for contract execution.

Finally, the transformed contract was deployed to the Ethereum blockchain using a decentralized application (more commonly known as a DApp in Ethereum terminology).

4.4.8 SUMMARY

In this chapter, the software tools and technologies utilized to implement the proposed solution was elaborated followed by the important functionalities of the proposed solution. The steps followed to create the source-to-source compiler were explained in detail in each subsection. The base contract and the Synergy marketplace contract which was needed to build the specific Solidity contract was also introduced.

Chapter 5 – Results and Evaluation

5.1 Introduction

This chapter elaborates the obtained results, how they are evaluated and the success level of the proposed solutions. An improved proof-of-concept evaluation model has been proposed in here to evaluate the results. Section 5.2 highlights the results obtained at each phase. Section 5.3 elaborates the evaluation model and section 5.4 evaluates the results obtained in comparison to approaches in previous works.

5.2 Transformed Contracts

The frontend of the compiler is capable of generating the parse tree for any contract, irrespective of its complexity. The grammar was written in such a way that the composable nature of the Peyton Jones' CDL is preserved. Therefore, the parse tree of complex contracts showcases this composable nature as well.

5.2.1 Parse tree for a Basic Contract

The graphical representation of a parse tree can be obtained by ANTLR4's TestRig. Figure 5.1 shows the parse tree obtained from TestRig GUI for the 'one' contract in the Peyton Jones' CDL.

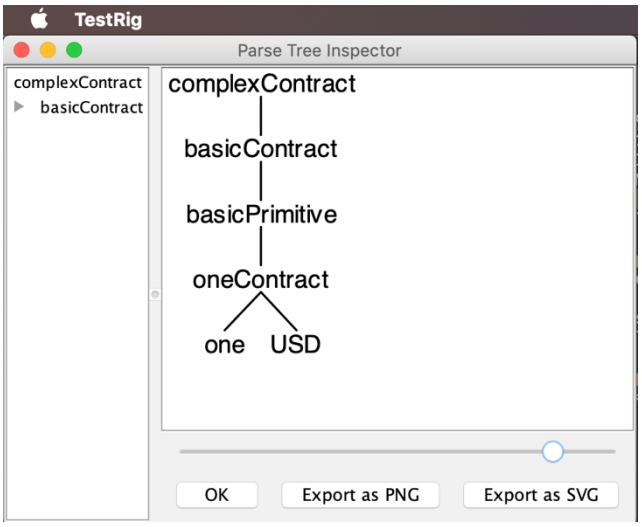


Figure 5.1 – One Contract TestRig output

Similarly, figure 5.2 shows the parse tree obtained for the Zero Coupon Bond, "receive $100 on the "10th of January 2019"".

Representation in Peyton Jones' CDL:

```
get ( truncate "10 Jan 2019" ( scale 100 ( one USD )))
```

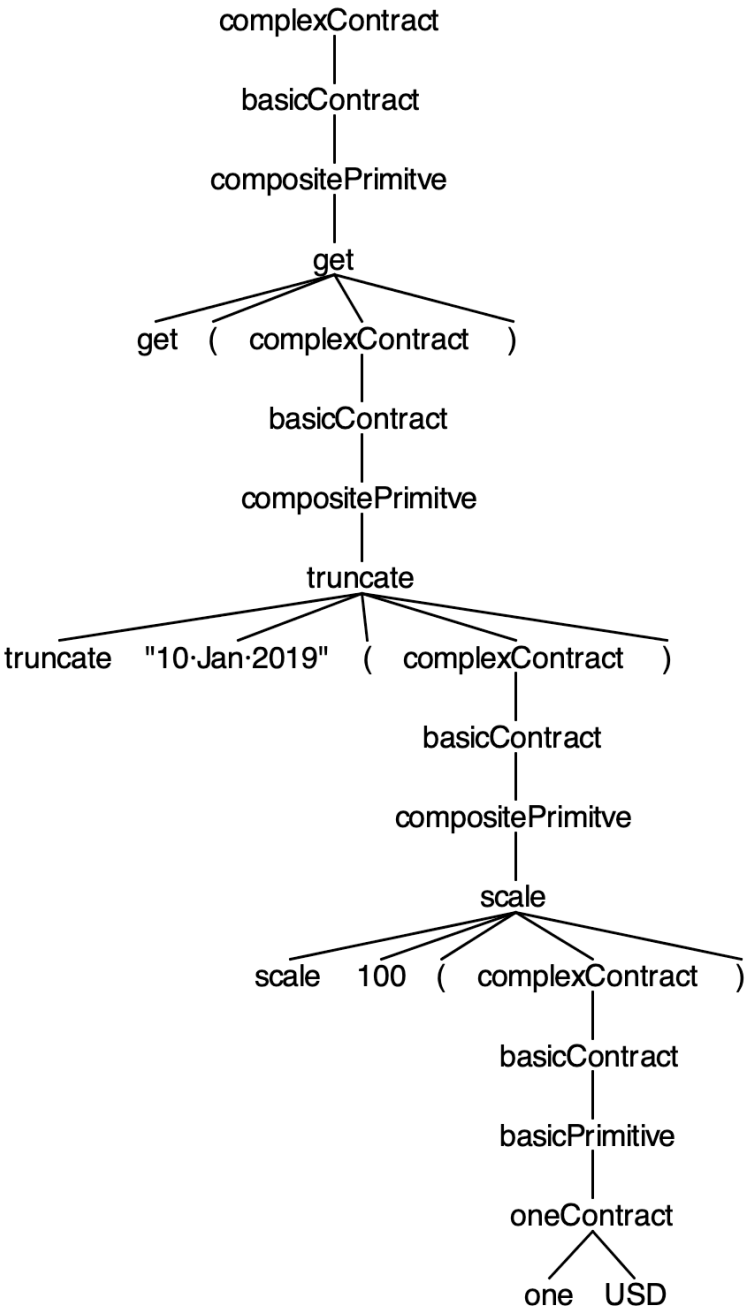


Figure 5.2 – Parse tree output of a Zero Coupon Bond

5.2.2 PARSE TREE FOR A COMPLEX CONTRACT

A complex contract is composed using basic contract components. Figure 5.3 shows the parse tree obtained from the compiler frontend for a complex contract.

Representation in Peyton Jones' CDL:

```
get ( scale 30 ( one GBP)) `and`  get ( truncate "t1" (scale 120 ( one GBP)))
```

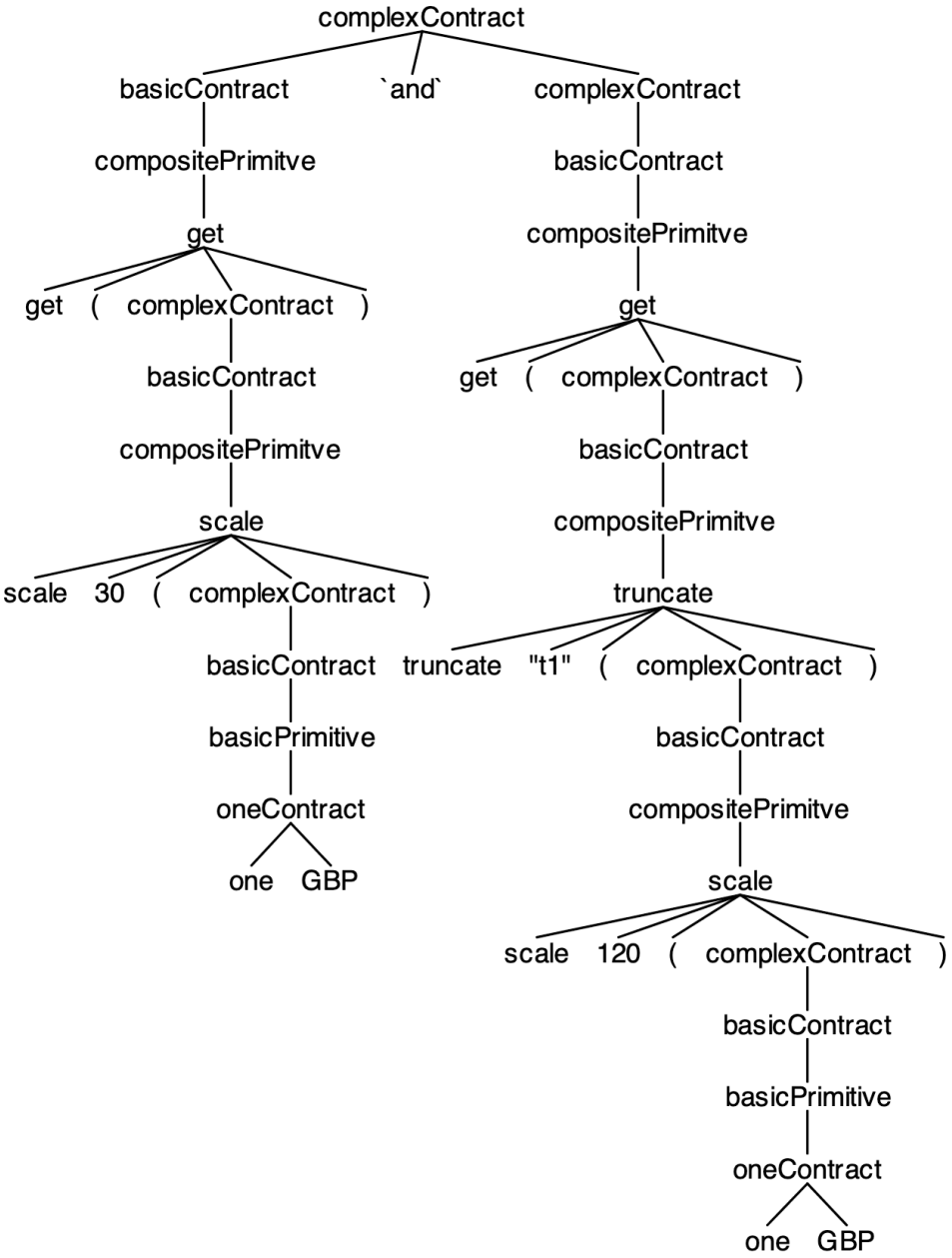


Figure 5.3 – Parse tree for a complex contract

5.2.3 Basic Contract in Solidity

Once the input of Peyton Jones' CDL is processed by the compiler, the transformed contract in Solidity was obtained. The contract was generated by the tree walker after walking the AST. The below code shows the Solidity version of the basic 'one' contract, where 1 GBP is immediately received at contract execution.

```
pragma solidity ^0.4.21;
pragma experimental ABIEncoderV2;
import {BaseContract, Marketplace} from './Marketplace.sol';

contract one is baseContract {
    constructor(Marketplace marketplace, string horizon, int value) public
BaseContract(marketplace, value, horizon) {
    }
    function proceed() public{
        marketplace_.receive(Marketplace.Commodity.GBP, 1);
    }
}

contract wrapper is baseContract {
    constructor(Marketplace marketplace, string horizon, int value) public
BaseContract(marketplace, value, horizon) {
    }
    function proceed() public{
        baseContract c = new c();
        marketplace_.get(c);
        c.proceed();
        kill();
    }
}
```

Contract invocation is done by calling the `proceed()` method in the wrapper contract which will instantiate the specific 'one contract' and call its `proceed()` function. The contract logic execution happens through this particular function.

Next, the Solidity contract code for a basic contract component (introduced before) is shown below. (i.e.: Receive $100 on a particular date in future)

```
pragma solidity ^0.4.21;
pragma experimental ABIEncoderV2;
import {BaseContract, Marketplace} from './Marketplace.sol';
contract c is baseContract {
   constructor(Marketplace marketplace, string horizon, int value) public
BaseContract(marketplace, value, horizon) {
   }
   function proceed() public{
       marketplace_.receive(Marketplace.Commodity.USD, 100);
   }
}
contract wrapper is baseContract {
   constructor(Marketplace marketplace, string horizon, int value) public
BaseContract(marketplace, value, horizon) {
   }
   function proceed() public whenAlive {
       baseContract c = new c();
       marketplace_.get(c);
       c.proceed();
       kill();
   }
}
```

5.2.4 Order of Execution

Figure 5.4 shows the order of execution of the Synergy framework. The process starts with the transformation of the Peyton Jones' contract and ends when the transformed contract logic is executed on the Ethereum blockchain.

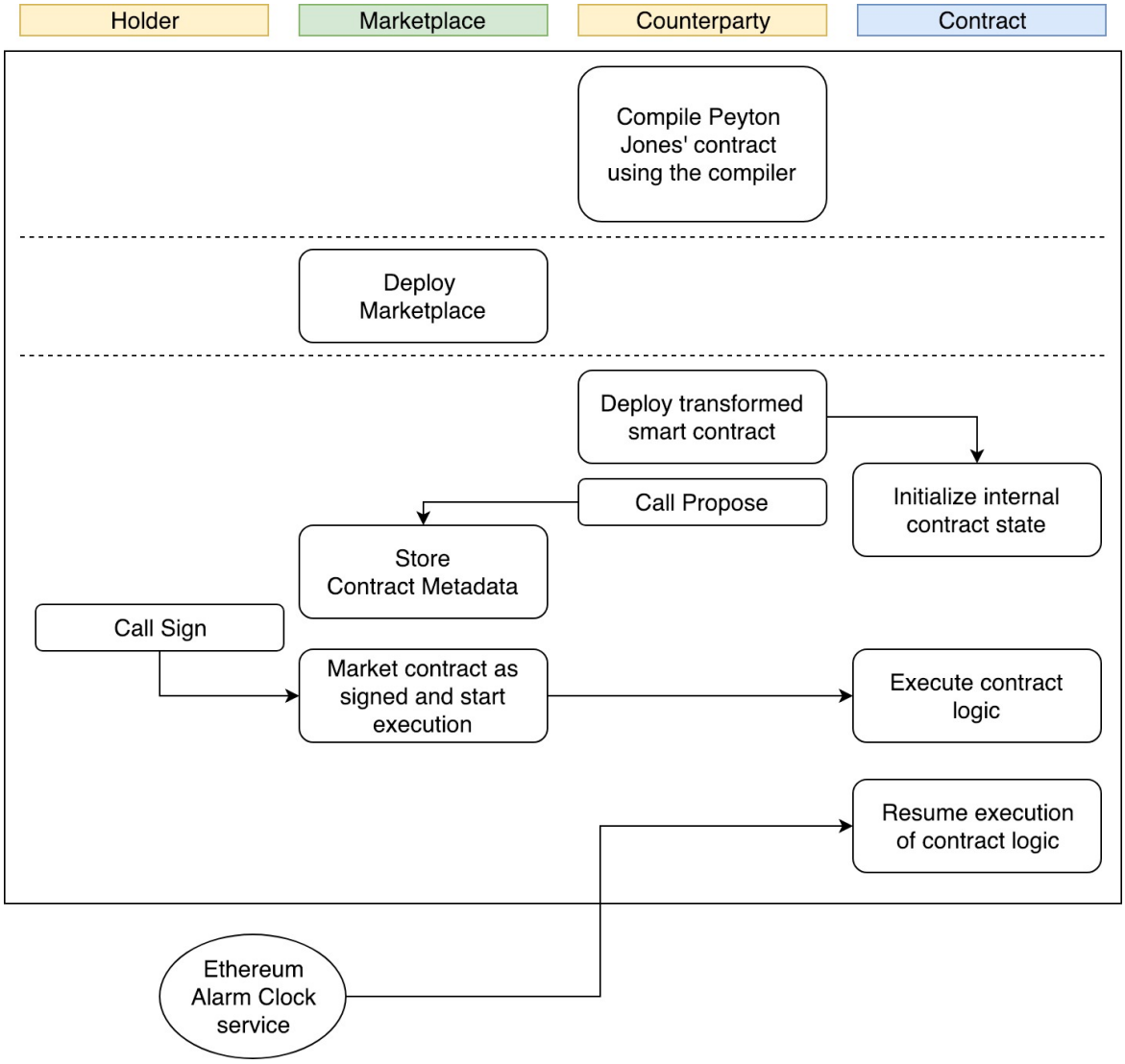


Figure 5.4 – Order of execution of a contract from transformation to contract logic execution

5.3 Evaluation

5.3.1 Evaluation Model

In order to evaluate the results obtained, the evaluation model was based on few key implementation choices and the success of those choices. Further, these choices were compared with three other alternatives; Merchant contracts [29], Findel contracts [16], traditional financial contracts.

The correctness of the contract was checked using the following model showed in figure 5.5.

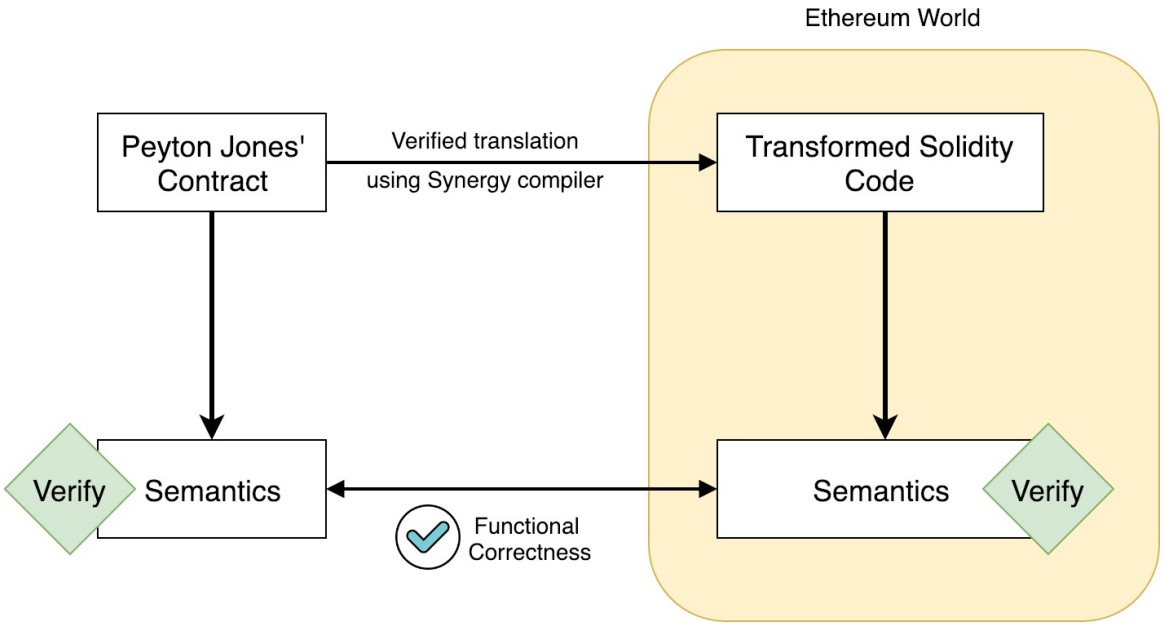


Figure 5.5 – Contract correctness evaluation

The semantic correctness of a contract was checked by comparing the operational semantics of an original Peyton Jones' contract with the operational semantics of the transformed Solidity version of the same contract. The transformed contract was executed and tested using Remix[4], the Solidity IDE.

5.3.2 Semantic Comparison

The contract semantics could be mapped out as follows. The comparison is done between a financial contract written in natural language, a financial contract represented in Peyton Jones' CDL and the contract which was transformed to a smart contract.

[4] **Remix** https://remix.ethereum.org/

Zero Contract

- *Natural language* – Do nothing.
- *Peyton Jones' CDL* – 'Zero'
- *Smart contract code snippet*

```
pragma solidity ^0.5.2;
pragma experimental ABIEncoderV2;
import {BaseContract, Marketplace} from './Marketplace.sol';

contract zero is BaseContract {
    constructor(Marketplace marketplace, string memory horizon, int scale) public
BaseContract(marketplace, horizon, scale) {}
    //No transaction performed as a result of this function
    function proceed() public alive{
        kill(BaseContract.KillReason.EXECUTED);
    }
}
contract wrapper is BaseContract {
    constructor(Marketplace marketplace, string memory horizon, int scale) public
BaseContract(marketplace, horizon, scale) {}

    function proceed() public alive{
        zero newContract = new zero(marketplace_, horizon_, scale_);
        marketplace_.get(newContract);
        newContract.proceed();
        kill(BaseContract.KillReason.EXECUTED);
    }
}
```

The *'zero'* contract is one of the most basic contracts in the financial contract world. It simply says to 'Do nothing'. Peyton Jones et al. have defined the *'zero'* combinator, in order to specify the contract which does nothing. The transformed Solidity code of this basic combinator does not execute any action (The proceed() function in the *'zero'* contract does not include any transaction). It simply kills the contract as soon as it is executed with no transaction happening. There is no action as a result of the contract. Therefore, the *'zero'* combinator and the *'zero contract in Solidity'* operates in the same manner which makes them semantically equivalent.

One Contract

- *Natural language* – Receive $1 immediately.
- *Peyton Jones' CDL* – 'one USD'
- *Smart contract code snippet* (complete contract code is included in the Appendix)

```
marketplace_.receive(Marketplace.Commodity.USD, 1);

function receive(Commodity commodity, int quantity) public {
     ContractMetadata storage c = contracts_[msg.sender];
     require(c.signed == true);
     balances_[c.counterparty][uint(commodity)] -= quantity;
     balances_[c.holder][uint(commodity)] += quantity;
}
```

The '*one'* contract is the other most basic contract among financial contracts. The semantics of this contract mentions that the counterparty should receive $1 immediately from the holder of the contract. Peyton Jones et al. have introduced the *'one'* combinator for this and the contract is written as *'one USD'*. When the transformed Solidity code of this contract is executed, the receive() function of the marketplace contract is called and it transfers $1 from the holder's account to the counterparty's account. Therefore, the behavior of the transformed contract is the same as how the semantics suggests in the Peyton Jones' contract. As such, the transformed *'one contract'* is semantically equivalent to the Peyton Jones' one contract.

Contract Component (i.e.: Zero Coupon Bond)

- *Natural language* – Receive $100 on the 31st of January 2019.
- *Peyton Jones' CDL* – 'get (truncate "31 January 2019" (scale 100 (one USD)))'
- *Smart contract code snippet*

```
marketplace_.receive(Marketplace.Commodity.USD, 100);
```

A basic contract component was introduced in this study in chapter 3, where it could have one or more combinators from *one, scale, truncate* and *get/give*. A zero coupon bond is one such contract which includes all these combinators. The semantics of this particular contract

states that $100 should be received by the counterparty at the specified future date. The transformed Solidity code of this particular contract transfers 100 units of USD to the counterparty on the said date when the propose() function of the wrapper contract is invoked. Therefore, the semantics stand correct for the basic contract component introduced as well.

Complex Contract (i.e.: A contract with operators and/or)

- *Natural language* – Receive $100 on the 31st of January 2019 `and` give $10 on the 5th February 2019.
- *Peyton Jones' CDL* – `'get ( truncate "31 January 2019" ( scale 100 ( one USD ))) `and` give ( truncate "5 February 2019" ( scale 10 ( one USD )))'`
- *Smart contract code snippet*

```
pragma solidity ^0.5.2;
pragma experimental ABIEncoderV2;
import {BaseContract, Marketplace} from './Marketplace.sol';

contract c1 is BaseContract {
    constructor(Marketplace marketplace, string memory horizon, int scale) public
BaseContract(marketplace, horizon, scale) {}
    function proceed() public alive {
        marketplace_.receive(Marketplace.Commodity.USD, 100);
        kill(BaseContract.KillReason.EXECUTED);
    }
}
contract c2 is BaseContract {
    constructor(Marketplace marketplace, string memory horizon, int scale) public
BaseContract(marketplace, horizon, scale) {}
    function proceed() public alive {
        marketplace_.receive(Marketplace.Commodity.USD, 10);
        kill(BaseContract.KillReason.EXECUTED);
    }
}
contract wrapper is BaseContract {
    constructor(Marketplace marketplace, string memory horizon, int scale) public
BaseContract(marketplace, horizon, scale) {
    }
    function proceed() public alive {
        c1 newContract1 = new c1(marketplace_, horizon_, scale_);
```

```
        c2 newContract2 = new c2(marketplace_, horizon_, scale_);
        marketplace_.get(newContract1);
        marketplace_.give(newContract2);
        newContract1.proceed();
        newContract2.proceed();
        kill(BaseContract.KillReason.EXECUTED);
    }
}
```

A complex contract has the nature of being combined by one or more operators (and/or). The semantics of a contract which has two basic contract components combined by 'and' implies that both basic contract components should be executed. The transformed Solidity code behaves in such a way, where both c1 and c2 contracts are executed on the blockchain. Each line in the Solidity code is executed sequentially which enables both c1 and c2 to proceed their contract logic. This replicates the functionality of the *'and'* combinator introduced by Peyton Jones et al. Therefore, two basic contract components tied together with an *'and'* behaves as expected in par with the representation of Peyton Jones' CDL.

Since the composable nature of contracts introduced in the Composing Contracts by Peyton Jones' et al. [2] is preserved through the compiler built in this study, it could be showed that if semantics are equivalent for the basic contract components and the operators of the transformed contracts, it is also equivalent for unforeseen complex contracts. The inductive logic of semantic equivalence was built as follows.

$$Base\ Case\ (k = 1)\ ;\ Basic\ Solidity\ Contract \equiv Basic\ Peyton\ Jones'\ Primitive$$

$$k\ ;\ Basic\ Solidity\ contract\ component \equiv Basic\ Peyton\ Jones' contract\ component$$

$$k + 1\ ;\ Basic\ Solidity\ contract\ `and/or`\ Basic\ Solidity\ contract$$
$$\equiv Basic\ Peyton\ Jones'\ contract\ `and/or`\ Basic\ Peyton\ Jones'\ contract$$

$$\therefore k = n\ ;\ Peyton\ Jones'\ Contract\ combined\ with\ `n`\ operators$$
$$\equiv Solidity\ Contract\ combined\ with\ `n`\ operators$$

Therefore, it was concluded that the transformed contract behaves as expected by the Peyton Jones' CDL representation and that the transformed contracts are semantically equivalent to Peyton Jones' contracts.

5.3.3 Evaluation of Implementation Choices

The implementation choices made in this study and certain features has been compared with one previous work [16] (referred to as *Findel*) and one parallel work [29] (referred to as *Merchant*). Implementation choices of this study are stated under *'Synergy'*, the name of the compiler for this study.

1. **Contract transformation output**

Table 5.1 shows the comparison of three implementations in terms of how the Peyton Jones' contract transformation to Solidity is achieved.

Table 5.1 – Contract transformation output comparison

	Findel	*Merchant*	*Synergy*
Choice	Only the marketplace contract is deployed. Creation of Findel contracts happen by calling functions in the marketplace contract.	A Solidity contract is created at each combinator which results in a nested series of contracts for a single input. The marketplace is deployed separately along with the contracts created for combinator.	The marketplace contract is deployed separately which includes common functions required for specific contracts. A new contract is created only at a *get/give*, *and/or* combinator. Information about the specific contract is retrieved and stored by the tree walker from the parse tree of the contract input, up until that point.
Evaluation	No need for a special-purpose compiler. Findel language was introduced and it is a deep embedding in Solidity.	A contract has been considered as a NFA and each combinator as a state of the NFA. As transition to each state requires a creation of a new contract, the efficiency is very low. (Contract creation is the single most expensive	The approach has been designed in a way to keep the number of contracts created at a minimal. For a contract that has one operator, only 3 specific contracts will be created in here; two for the basic contract components and one for the operator. Therefore, this method is much efficient opposed to

		operation on Ethereum – costs 32000 gas)	creating a nested series of contracts such as in Merchant [29].

2. Autonomous execution and execution guarantee

Table 5.2 shows the comparison of three implementations in terms of whether contracts have autonomous execution and whether there is an execution guarantee for contracts.

Table 5.2 – Autonomous execution and execution guarantee comparison

	Findel	*Merchant*	*Synergy*
Choice	Findel is a language embedded in Solidity. Therefore, inherently Findel contracts are smart contracts. However, Findel contracts are not guaranteed to execute.	Contract enforcement should be initiated externally. There is no autonomous execution. If a contract is not enforced, it is considered as a mutual agreement to void the contract.	Contract enforcement happens autonomously through the Ethereum alarm clock (EAC) service. The timeline of the contract is not required to be tracked as the EAC service monitors the horizons of the contracts and executes at the correct date.
Evaluation	If there is no execution guarantee, the timeline of the financial contracts would not be functional. Therefore, this is a limitation in the system.	One of the key objectives of autonomous execution is not met. Explicit invocation is not very feasible. Therefore, there is no execution guarantee as well.	EAC is an incentive based service. The users will be given an incentive for invoking a call to the contract. This is initiated by the service itself. This meets the objective of autonomous contracts in a trustless environment. However some preprocessing is required such as pre-payment of gas, incentives to the random user who will be doing the invocation, etc.

3. Execution cost

The execution cost on the Ethereum blockchain is measured in units of gas on the Ethereum blockchain. A gas estimation is done beforehand and then adequate number of gas units are specified for a contract. If the execution runs out of gas before completion, the contract gets void and the gas is lost. The executor cannot regain the gas spent. Therefore, it is critical to specify adequate number of gas units for execution completion.

Gas price is very competitive on the Ethereum blockchain among the peers. As of January 2019, the price of one Ether is about $158. Additionally, the minimum cost per unit of gas is approximately 1.1 Gwei (2 x 10^{-9} Ether). The cost to deploy the Findel [16] marketplace, for example, is calculated as follows;

$$2 \times 10^{-9} \ \frac{Ether}{gas} \times \frac{\$\,158}{Ether} \times 1797270\ gas = \$\,0.568$$

Remix, the in-browser Solidity compiler and blockchain simulator was used in this study to compile, deploy and measure the contracts. All contracts were compiled with the Solidity compiler `--optimize` flag enabled. Remix reports both transaction cost and execution cost.

Figure 5.6 shows the comparison of transaction costs of different actions of Findel contracts, Merchant contracts and Synergy contracts.

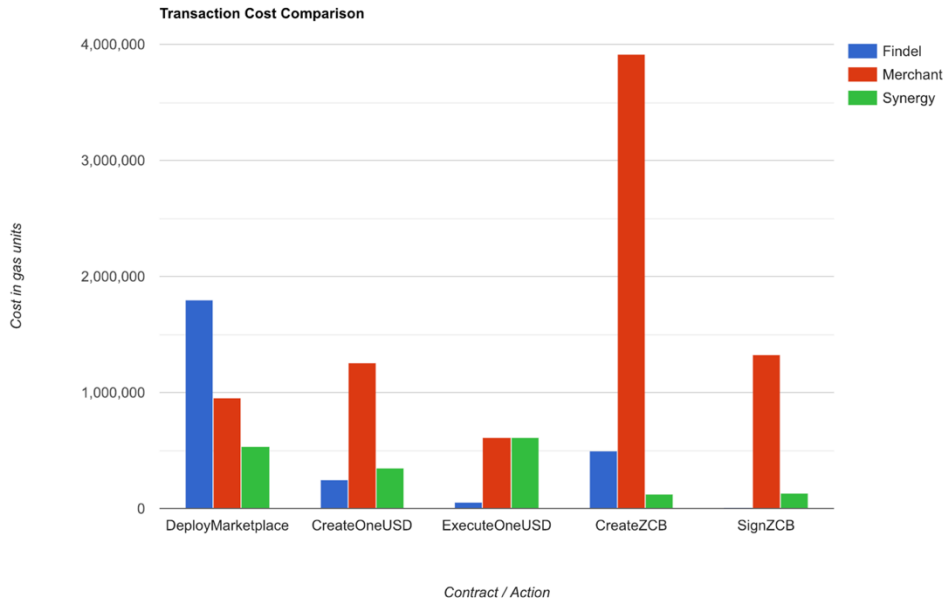


Figure 5.6 – Comparison of transaction costs in terms of gas units on the Ethereum blockchain

The graph indicates that Synergy contracts are more efficient in terms of creation of specific contracts, executing different actions, etc. than the other two contract frameworks. The main reason for the low transaction cost of contract creation in Synergy is due to the fact that minimal number of contracts being created compared to Merchant. This makes the overall execution efficient and less costly for the owners of contracts.

5.4 Synergy over Traditional Financial Contracts

The main objective of this study was to facilitate autonomous execution of a financial contract in a trustless environment without the interference of a CCP. Traditional financial contracts suffer from the fundamental risk of not being executed at the correct date and also the parties in the contract have to trust the CCP in order to give them the correct result. Therefore, the main motivation behind this study was to map two domains (financial contract domain and the smart contract domain) to eliminate the CCP and achieve better contract execution. Therefore, the proposed solution in this study has the following advantages over traditional contracts.

- Autonomous execution; no need of tracking a contract timeline.
- Execution in a trustless environment; a CCP is not required.
- Less errors than handcrafting manual contracts.
- Time-to-market is low because complex contracts are composed using basic contracts than being written from scratch.

5.5 SUMMARY

This chapter elaborated on the proposed evaluation model for contract transformation from the financial contract domain to the smart contract domain. The results obtained were explained in detail and an evaluation framework was proposed. The Synergy contracts of this study were compared with other approaches in the past and also with a parallel work and the benefits of the proposed solution was highlighted over the others. Finally, the latter part of this chapter stated how the main objectives of this study has been achieved and why it contains a benefit over traditional financial contracts.

Chapter 6 – CONCLUSIONS

6.1 INTRODUCTION

This chapter includes a review of the research aims and objectives, research problem, limitations of the current work and implications for further research.

6.2 CONCLUSIONS ABOUT RESEARCH QUESTIONS (AIMS/OBJECTIVES)

The main aim (included in the first research question) of this study was to eliminate the CCP of financial contracts in order to reduce execution risks of financial contracts. In order to achieve this aim, the opportunity was captured with the emergence of Ethereum and smart contracts, where autonomous execution in a trustless environment was possible. The proposed solution to achieve the objective in this study consisted of few steps. The Peyton Jones' CDL was extended to a deep embedding, a source-to-source compiler was built in order to convert financial contracts to smart contracts, enhanced the compiler to preserve the composable nature of the Peyton Jones' CDL and finally the converted contracts were deployed to the Ethereum blockchain to check their functionality. We identified that even though there were few semantic differences between the Peyton Jones' CDL and Solidity, it was possible to build a compiler which performed the contract conversion successfully. Through this, we managed to achieve the main objective of this study.

The second research question was about preserving the properties of the Peyton Jones' CDL in the transformed contract. Through the evaluation framework proposed in this study, we identified that the semantics were preserved in the transformed contract. Further, the composable nature of the Peyton Jones' contracts was also preserved through the compiler built in this study as the transformed contract too portrayed composability. It was possible to let the transformed contract execute at a future date through the Ethereum Alarm Clock service which guaranteed autonomous execution at a future date. An initial objective of this

study was to explore the contract calendar of a transformed contract. A dynamic contract calendar was built in a previous work [21] for traditional financial contracts. Even though it was highlighted in this paper, that a contract calendar is a must to observe the dynamic nature of contracts, we identified that a contract calendar is of no use for the contract execution in our solution. This is due to the fact that it facilitates autonomous execution as a result of the financial contract being a smart contract. Thus, it can be concluded that the proposed solution in this study is a feasible mechanism to eliminate the CCP of financial contracts and facilitate autonomous execution in a trustless environment.

6.3 CONCLUSIONS ABOUT RESEARCH PROBLEM

The risks of central counterparty involvement in financial contracts has been a major concern. The need was out there to build autonomous contracts which did not need a central counterparty to manage the transactions. There were many attempts in the past to re-implement financial contracts in the Ethereum blockchain domain [16] [24] . However, they suffered from certain issues due to the error-prone nature of Solidity and Ethereum platform limitations. After much research on the domain, we identified that a financial contract is best represented using a DSEL. Therefore, re-implementing it on the smart contract domain did not seem beneficial as it will eliminate important properties of the DSEL. Therefore, a solution which was more beneficial was to transform the contract from one domain to the other. There were no proper implementation in converting a financial contract to a smart contract in past researches. This was a clear research gap identified after conducting the literature review.

This study has contributed to the financial contract domain significantly as it reduces certain risks imposed on contracts due to manual traditional methods that are being used in the present context. For example, the proposed solution of this study was able to eliminate the CCP risk, mitigate counterparty risk and credit risks. This improves the reliability and efficiency of contracts drastically. Further, having autonomous execution would eliminate the need for the counterparties to keep track of the contract calendar manually.

A source-to-source compiler was built in this study which is a significant computer scientific contribution. The compiler can be extended to transform contracts to any suitable scripting language. Solidity was chosen in here as it is the most used and tested scripting language for the Ethereum blockchain. However, it is just an intermediate language for the transformation purpose.

With the contributions of this study, it could be concluded that smart contracts could be the first step to formulate a reliable, fast and transparent financial market. Having shown that financial contracts can be represented on a decentralized network such as the blockchain, this study takes one step closer to the reality of a transparent, trustless financial market in future.

In summary, this study has proposed a source-to-source compiler to convert financial contracts to smart contracts, explored the functionality of such transformed contracts when executed on the Ethereum blockchain and has proven that financial constructs suggested by Peyton Jones et al. can be easily expressed on a blockchain. Overall, it was shown that it is possible to achieve autonomous financial contracts to execute in a trustless environment making the CCP involvement insignificant for contract execution.

6.4 LIMITATIONS

The language extension done in this study was only for the combinators of contracts. The combinators for observables weren't extended nor utilized for the transformation in the compiler. Further, the study was restricted to the Peyton Jones' CDL as the source language and did not explore any other DSEL for financial contracts due to time constraints. A readily usable, optimized compiler was not built through this study, but only a workable version of the compiler was developed due to time constraints.

The extent to which a transformed contract is reliable, transparent and efficient depends on the facilitations by the Ethereum platform as well. Since the Ethereum platform and Solidity

is still under development, certain limitations are imposed on our results as well. Few such limitations are discussed below.

- **Reliability** of contract execution is mostly weighed upon the Ethereum platform. One major dependency of contract execution in this study was facilitating autonomous execution through the Ethereum Alarm Clock service. This is a third party service which needs to be integrated explicitly as a separate smart contract. The lifetime of autonomous execution of contracts proposed in this study depends on how long this service would be maintained by the developers/community.

- **Efficiency** is another major problem when it comes to financial contracts. Financial markets operate on the scale of milliseconds [18]. However, operations on the blockchain cannot accommodate such speed efficiency. As of now, it takes 15 seconds approximately for a block to be mined and to be added to the blockchain and the maximum throughput of the Ethereum blockchain is approximately 15 transactions per second. By contrast, the NYSE Group processes around 200 trades per second. Therefore, the blockchain does not seem to facilitate the speed required by financial markets. However, this is still an ongoing area of research [19].

- **Transparency** is achieved on the blockchain by the distributed ledger system. Each peer on the network would be aware of all transactions and contract executions on the blockchain. However, since Ethereum addresses are not tied up with public identities, transparency in terms of who performs a transaction is limited. This creates a problem of liability on the network in terms of debt enforcement when executing contract logic.

Even though these limitations have minor impact on our proposed solution, further research and development is required to mitigate these in order to use such a system in a real financial market.

6.5 Future Work

6.5.1 Compiler Improvements

The correctness of the compiler needs to be proved before using Synergy in a live environment. Popular DApp frameworks such as Truffle or Embark could be used to build a test suite for contracts as they include a test framework as well. Further, the compiler could be extended to support observables. This is a key extension that will be required as most derivatives execute based on observable values. The compiler could be optimized even more in future to reduce the number of contracts created for a particular input. This would reduce execution costs and increase performance of contract execution on the blockchain.

6.5.2 Debt Enforcement

The proposed solution does not state the consequences of executing a contract when the holder's commodity balance is zero. In such a situation, there would be no commodities to transfer to the counterparty when the contract is executed, resulting in addition of debt for the holder. How debt is handled on the Ethereum blockchain remain a complex and unsolved issue. This would require extensive further work in order to come up with a solution.

6.5.3 Reducing Compiler Overhead

The proposed solution in this study required the contracts to be compiled to an intermediary language, Solidity. However, a more optimized solution would be to directly compile contracts to the Ethereum Virtual Machine bytecode from the Peyton Jones' CDL. This would reduce the overhead in compilation. This would require significant development effort in the future.

REFERENCES

[1] B. K. a. R. Martin, "Decentralized versus centralized financialsystems: is there a case for local capitalmarkets?," *Journal of Economic Geography 5,* no. Advance Access published on 22 June 2005 , p. 387–421, 2005.

[2] J.-M. E. J. S. Simon Peyton Jones, "Composing contracts: an adventure in financial engineering (functional pearl)," in *ICFP '00 Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, 2000.

[3] "Structuring, pricing and processing complex financial products with MLFi, Tech. Rep.," [Online]. Available: http://www.lexifi.com/files/resources/MLFiWhitePaper.pdf. [Accessed May 2018].

[4] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[5] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2013.

[6] "Ethereum for dummies," Coinmama, 14 March 2016. [Online]. Available: https://www.coinmama.com/blog/ethereum-for-dummies. [Accessed 2018 April].

[7] N. Szabo, "The idea of smart contracts," 1997.

[8] "Solidity Documentation," [Online]. Available: https://solidity.readthedocs.io/en/v0.4.24/. [Accessed May 2018].

[9] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014. [Online]. Available: http://gavwood.com/Paper.pdf. [Accessed May 2018].

[10] D. I. J. M. Gaillourdet, "A software language approach to derivative contracts in finance," *CEUR Workshop Proceedings,* vol. 750, pp. 39-43, 2011.

[11] A. Mediratta, "A Generic Domain Specific Language For Financial Contracts," Graduate School—New Brunswick Rutgers, 2007.

[12] N. Szabo, "A Formal Language for Analyzing Contracts.," 2002. [Online]. Available: https://web. archive. org/web/20160810220820/http://szabo. best. vwh. net/contr actlanguage.html . [Accessed March 2018].

[13] D. C. W. K. M. B. R. A. M. J. B. A. I. …. B. M. Mills, "Distributed Ledger Technology in Payments, Clearing, and Settlement. SSRN.," 2016.

[14] J. a. E. R. Pettersson, "Safer smart contracts through type-driven development.," Chalmers University of Technology, Göteborg, 2016.

[15] S. S., "Decomposing contracts," University of Bonn, 2014.

[16] A. K. D. &. T. S. Biryukov, "Findel: secure derivative contracts for Ethereum," *International Conference on Financial Cryptography and Data Security,* pp. 453-467, 2017.

[17] J. G. a. O. d. Moor, "Composing Contracts," in *The fun of programming*, Palgrave Macmillan, 2003.

[18] S. L. P. J. a. J.-M. Eber, "How to write a financial contract," in *The Fun of Programming,*, 2003.

[19] E. E. F. H. J. G. S. a. C. S. J. Andersen, "Compositional specification of commercial contracts," *International Journal on Software Tools for Technology Transfer (STTT),* vol. 8, no. 6, p. 485–516, 2006.

[20] C. &. N. C. Keppitiyagama, "Domain specific language for specifying operations of a central counterparty," in *Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer)*, 2017.

[21] C. I. K. K. G. G. V. R. Balalla, "Use of Peyton Jones' Contract Descriptive Language to Evaluate Different Value Processes," 2017.

[22] C. Dannen, Introducing Ethereum and Solidity., Berkeley: Apress., 2017.

[23] S. C. J. S. V. A. M. Eskandari, "On the feasibility of decentralized derivatives markets," 2017.

[24] B. E. M. H. F. R. O. Egelund-Müller, "Automated Execution of Financial Contracts on Blockchains," *Business and Information Systems Engineering,* 2017.

[25] C. Zhou, "A source-to-source compiler for the PRAM language Fork to the REPLICA many-core architecture," Linköping University, 2012.

[26] "ANTLR," [Online]. Available: http://www.antlr.org. [Accessed October 2018].

[27] R. G. A. &. G. M. Boulton, "Experience with embedding hardware description languages in HOL," in *IFIP*, 1992.

[28] G. Wood, "LLL PoC 6," 2014. [Online]. Available: https://github.com/chriseth/cpp-ethereum/wiki/LLL-PoC-6. [Accessed March 2018].

[29] R. Gardiner, "Free Trade: Composable Smart Contracts," University of Bristol, 2018.

APPENDIX A – CODE LISTINGS

The following code segment shows the implementation of the tree walker of the compiler.

```
function recurse (tree, list) {
    let tempList = [];
    let currentKeyword;

    if (tree.getChildCount() == 0) {
        let nodeText = tree.getText();
        if (nodeText === "(" || nodeText === ")") {
            return list;
        }
        list.push(tree.getText());
        //console.log(list);
        return list;
    }

    for (let i = 0; i < tree.getChildCount(); i++) {
        tempList = recurse(tree.getChild(i), tempList);
    }

    for (let i = 0; i < tempList.length; i++) {
        let keywordList = ['get', 'scale', 'one', 'zero', 'give', 'truncate', 'then',
'anytime', '`and`', '`or`'];
        for (let j = 0; j < keywordList.length; j++) {
            if (keywordList[j] === tempList[i]) {
                currentKeyword = tempList[i];

            }
        }
    }

    switch (currentKeyword) {
        case 'scale': {
            list.push(tempList[1]);
            list.push(tempList[2]);
            return list;
        }
        case 'get': {
            let commodity = tempList[1].contractValue[1];
```

```
            let quantity = tempList[1].contractValue[0];
            let contractName = "c";
            let contract = `` +
                `contract ${contractName} is BaseContract {\n` +
                `    constructor(Marketplace marketplace, string memory horizon, int
scale) public BaseContract(marketplace, horizon, scale) {\n` +
                `    }\n` +
                `    function proceed() public alive{\n` +
                `        marketplace_.receive(Marketplace.Commodity.${commodity},
${quantity});\n` +
                `        kill(BaseContract.KillReason.EXECUTED);\n`+
                `    }\n` +
                `}\n`;

            fs.appendFile("./contractFiles/test.sol", contract, function (err) {
                if (err) {
                    return console.log(err);
                }

                console.log("The file was saved!");
            });
            list.push('gt');
            return list;
        }
        case 'give': {

            let contractName = "c";
            let commodity = tempList[1].contractValue[1];
            let quantity = tempList[1].contractValue[0];
            let contract = `` +
                `contract ${contractName} is BaseContract {\n` +
                `    constructor(Marketplace marketplace, string memory horizon, int
scale) public BaseContract(marketplace, horizon, scale) {\n` +
                `    }\n` +
                `    function proceed() public alive{\n` +
                `        marketplace_.receive(Marketplace.Commodity.${commodity},
${quantity});\n` +
                `        kill(BaseContract.KillReason.EXECUTED);\n`+
                `    }\n` +
                `}\n`;


            fs.appendFile("./contractFiles/test.sol", contract, function (err) {
```

```
                if (err) {
                    return console.log(err);
                }

                console.log("The file was saved!");
            });
            list.push('gv');
            return list;
        }
        case 'truncate': {
            let truncateObj = {};
            truncateObj.horizon = tempList[1];
            truncateObj.contractValue = [tempList[2], tempList[3]];
            list.push(truncateObj);
            return list;
        }
        case 'one': {
            return [tempList[1]];
        }
        default: {
            if(list.length > 0){
                for(let i=0; i<tempList.length;i++)
                    list.push(tempList[i]);
                return list;
            }
            return tempList;
        }
    }
}
```

The following code segment shows the implementation of the Solidity contract creation.

```
function contractCreation(tree) {
    let list = [];
    let contractName = "wrapper";
    let headers = `` +
        `pragma solidity ^0.5.2;\n` +
        `pragma experimental ABIEncoderV2;\n` +
        `import {BaseContract, Marketplace} from './Marketplace.sol';\n`;

    fs.writeFileSync("./contractFiles/test.sol", headers, function(err) {
        if(err) {
```

```
            return console.log(err);
        }

        console.log("The file was saved!");
    });
    let contract = ``;
    list = recurse(tree, list);
    console.log(list);
    if( list[0] === 'gt'){
        contract = contract + `contract ${contractName} is BaseContract {\n` +
            `   constructor(Marketplace marketplace, string memory horizon, int
scale) public BaseContract(marketplace, horizon, scale) {\n` +
            `   }\n`+
            `   function proceed() public alive{\n` +
            `       c newContract = new c(marketplace_, horizon_, scale_);\n` +
            `       marketplace_.get(newContract);\n` +
            `       newContract.proceed();\n` +
            `       kill(BaseContract.KillReason.EXECUTED);\n`+
            `   }\n` +
            `}\n`;
    }
    else if(list[0] === 'gv'){
        contract = contract + `contract ${contractName} is BaseContract {\n` +
            `   constructor(Marketplace marketplace, string memory horizon, int
scale) public BaseContract(marketplace, horizon, scale) {\n` +
            `   }\n`+
            `   function proceed() public alive{\n` +
            `       c newContract = new c(marketplace_, horizon_, scale_);\n` +
            `       marketplace_.give(newContract);\n` +
            `       newContract.proceed();\n` +
            `       kill(BaseContract.KillReason.EXECUTED);\n`+
            `   }\n` +
            `}\n`;
    }
    console.log(contract);
    fs.appendFile("./contractFiles/test.sol", contract, function (err) {
        if (err) {
            return console.log(err);
        }

        console.log("The file was saved!");
    });
}
```