Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

An Embedded Domain Specific Language in Haskell for Generating Smart Contracts

Darren Kitching

April 28, 2021

Supervisor: Dr. Andrew Butterfield

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
B.A. (Mod.) Computer Science

Declaration

I hereby declare that this Final Year Project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

Smart contracts are an emerging application of blockchain technology which are capable of fundamentally changing how transactions are performed in the real world. The greatest barrier to more widespread adoption of smart contract technology is the need to learn new programming languages specialised to the task of writing smart contracts such as the Solidity programming language. There are also security concerns associated with smart contracts as bugs in the code can lead to the theft of funds or a lack of understanding of the code by one party to a contract may lead to their exploitation.

This project seeks to make smart contract creation an easier process by removing the barrier of having to learn an entirely new programming language. This project achieves this by implementing an embedded domain specific language in the Haskell programming language which allows the user to write safe and secure smart contracts in Haskell that can then be compiled to Solidity code and run on the Ethereum Virtual Machine. By allowing smart contract creation in a popular general-purpose programming language this project seeks to open the possibility of smart contract design to the broader programming community. This project also seeks to investigate the suitability of Haskell as a host language when implementing embedded domain specific languages.

Acknowledgements

I would like to thank my family for their unwavering support throughout the last four years. They were always there for me during both the highs and lows. I know I would not have made it this far without them.

I would also like to thank my project supervisor, Andrew Butterfield, for his guidance and support these past eight months.

Lastly, my thanks to my friends for all of the laughs along the way. They made the whole experience worth while and kept me sane through a long year of lockdown.

Contents

List of Figures

List of Tables

Listings

Terminology

DAO:	Decentralized Autonomous Organisation
DSL:	Domain Specific Language
eDSL:	Embedded Domain Specific Language
Ether:	Ethereum's native cryptocurrency
Ethereum:	An open source decentralised blockchain
EVM:	Ethereum Virtual Machine
Gas:	The fee required to do work on the Ethereum blockchain
Smart Contract:	A self-executing contract
Wei:	The smallest denomination of Ether valued at one quintillionth of an Ether
Zcb:	Zero-coupon bond

1 Introduction

1.1 Context of Problem

Smart contracts are generally composed in specialised programming languages such as the Solidity programming language which was designed for writing smart contracts deployable on the Ethereum Virtual Machine. This means that in order to write safe and reliable smart contracts, programmers must be knowledgeable in one of these programming languages which is a large barrier to entry to begin composing smart contracts. It also means that smart contracts are not easily understandable to people who do not possess a deep understanding of these specialised languages and therefore smart contracts are rarely used outside of the field of Computer Science where they would be most useful such as in business and the legal profession. A lack of understanding of how smart contracts work can allow for the exploitation of contract participants. Similarly, unintentional bugs in smart contracts can lead to exploits resulting in theft of large sums of cryptocurrency.

1.2 Motivation

The motivation for undertaking this project was to make the creation of smart contracts more accessible to programmers who do not have specific knowledge of programming languages built for smart contract composition. The project was also motivated by a desire to make smart contracts more understandable to people who do not necessarily have a background in computer science or programming by allowing smart contracts to be composed in a human readable syntax. By allowing smart contracts to be composed in a more understandable syntax, it is hoped that no party to the contract would be exploited by misunderstanding the terms of the contract being agreed to. The project was also motivated by a desire to provide an easy way of composing secure smart contracts by ensuring that the contracts created provide secure transactions where funds are concerned. Finally, the project was motivated by a desire to explore domain specific language creation in a functional programming language and to justify Haskell's reputation as a good host language for embedded domain specific languages.

1.3 Project Objectives

The objective of this project is to design and implement an embedded domain specific language in Haskell that is capable of generating smart contracts deployable on the Ethereum Virtual Machine. The domain specific language created should use a simple to understand syntax to allow people that do not have specialised knowledge of smart contracts to quickly compose them. The domain specific language created should be powerful, with the ability to compose many useful smart contracts. The language should ensure that the generated smart contracts are secure to the greatest possible extent and bug free. The final objective of this project is to examine Haskell's suitability as a host language for embedded domain specific languages by using Haskell's language features to solve the project objectives.

1.4 Outline of the Project Work

The project first involved studying the Solidity programming language and widely used smart contracts to gain an understanding of commonly used smart contract functionality and its corresponding representation in Solidity. The project then involved designing a domain specific language syntax capable of representing useful smart contracts in a simple and human readable manner. The domain specific language was then implemented in Haskell along with a way of compiling contracts written in the domain specific language to valid Solidity syntax so that contracts could be compiled using the Solidity compiler and run on the Ethereum Virtual Machine. Example contracts were then implemented in the DSL to demonstrate its functionality and identify any shortcomings that could be rectified. Finally, tests were written for all of the DSL example contracts implemented to ensure all functionality was working as expected when a contract was deployed.

1.5 Outline of the Report

1.5.1 Background

This chapter will begin by providing a brief description of the Haskell programming language and explain why it is often considered a good host language for embedded domain specific languages. It will then provide details of Ethereum, smart contracts and the Solidity programming language. An overview of financial combinators will be provided and finally the chapter will look at the emerging blockchain platform Cardano and discuss other attempts at smart contract DSLs.

1.5.2 Design

This chapter will lay out a design framework for a domain specific language to be built in Haskell which can generate Solidity smart contracts. The structure of the domain specific language and the features that it will provide will be described as well as the justification for the design decisions taken.

1.5.3 Implementation

This chapter will walk through the implementation of the domain specific language in the Haskell programming language. It will detail the various steps involved in building a domain specific language that is capable of outputting smart contracts represented in valid Solidity syntax so that they can be compiled with the Solidity compiler and deployed on the Ethereum Virtual Machine.

1.5.4 Evaluation

This chapter will evaluate the domain specific language built by discussing how the language was tested and any shortcomings the language has. A comparison between the final domain specific language built and what was specified as the project's objectives will be made to determine how many of the established goals were reached.

1.5.5 Conclusions

This chapter will provide a brief reflection on the process of doing this project and discuss how the project may be useful in future work related to this field.

2 Background

2.1 Haskell

2.1.1 Haskell Overview

Haskell is a general-purpose functional programming language. Many of Haskell's design features such as purity, lazy evaluation and a powerful type system make it useful for the implementation of embedded domain specific languages. The pattern matching structure Haskell uses is also well suited to the representation of language grammars in Haskell as it provides an easy way to translate from language rules to a Haskell implementation.

2.1.2 Haskell Purity

Haskell is a pure programming language meaning that it does not allow side effects (1). This means that a function being run with the same parameters in Haskell will always give the same output. This is not always guaranteed in impure programming languages where variables outside of a function's parameters can affect a function's output.

2.1.3 Haskell Lazy Evaluation

Haskell uses lazy evaluation meaning that the value of an expression is not calculated until it is forced. This allows certain code to be run in Haskell that would loop infinitely in a language with strict evaluation.

```
1 count_from_six :: [Integer]
2 count_from_six = enumFrom 6
3
4 main = putStrLn $ show $ take 5 $ count_from_six
```

Listing 2.1: An example of Lazy evaluation in Haskell.

Listing 2.1 shows a program that would not evaluate in a programming language with strict evaluation. The function count_from_six creates an infinite list of integers starting from six and increasing by one each entry. The main function is then printing the first five entries from that list to the console. In a strict programming language,

the full list created by count_from_six would have to be evaluated before 'take 5' can act on it. This would mean the program would try to evaluate an infinite list and would therefore loop infinitely until it eventually ran out of memory. In a lazy language such as Haskell however, only expressions that are immediately needed are evaluated. Printing 'take 5' will only require the first five elements of count_from_six to be evaluated so the program outputs correctly as [6, 7, 8, 9, 10].

2.1.4 Haskell Type System and Pattern Matching

Haskell is a strictly typed language which is ideal for domain specific language creation as it ensures correctness throughout the program in order to compile successfully. Haskell supports ad-hoc polymorphism meaning that polymorphism is possible but an implementation for each class instance that is used must be given. Types are defined in Haskell using the 'data' keyword and type synonyms are implemented using the 'type' keyword. Recursive data types are possible when the keyword in the left-hand side of the data constructor also appears on the right hand side as an argument.

Haskell's pattern matching syntax provides a clear and concise way of representing language grammars in Haskell and this is useful for domain specific languages. Consider an example of the Peano Numbers which can be used to represent the natural numbers. A Peano number can either be zero or the successor of a Peano number. In Haskell the entire structure of the language grammar is represented succinctly in one line as seen in listing 2.2.

```
1 data Number = Zero | Succ Number
2
3 add :: Number -> Number -> Number
4 add (Zero) (b) = b
5 add (Succ (a)) (b) = Succ (add (a) (b))
```

Listing 2.2: A demonstration of Haskell's type system representing Peano Numbers.

The bar on the right hand of data Number specifies that the data type can be constructed either by using the constructor Zero or the constructor Succ with a Number. The 'add' function can then be implemented using pattern matching. The function takes two instances of Number and produces a Number. When the first argument is Zero, the function returns the second argument. When the first argument is the successor of some Number a, add is called on a and b and the return value from that call is wrapped in the Succ that has been removed by the pattern match. This pattern matching structure can be applied to complex language grammars in a

relatively straightforward way by having a pattern match for each path a rule can take.

2.1.5 Haskell's Maybe Type

Haskell's Maybe type is a powerful tool for specifying an optional element in a language grammar rule. Maybe can encapsulate a data type and can be used to specify that the data type is optional. For example, we create a data type called Name which contains two strings, one representing a person's first name and the second representing their last name. We may wish to make the last name component optional so someone can choose to just represent their first name instead of representing both first and last names. We accomplish this by making the first argument of the Name constructor a non-optional string but making the second argument of the Name constructor a Maybe string as seen in listing 2.3. If the data type is being used to represent just somebody's first name, then the second argument is set to Nothing. If the data type is being used to represent someone's first and last name, then the first argument is a string representing their first name and the second argument is a string representing their second name wrapped in a Just.

```
1 data Name = Name String (Maybe String)
2
3 john = Name "John" Nothing
4 patrickMurphy = Name "Patrick" (Just "Murphy")
```

Listing 2.3: An example of Haskell's Maybe type.

The Maybe type is also useful for incorporating error prevention into Haskell programs. The Nothing constructor can be used to represent a failed function call rather than simply throwing an error and stopping the program. This allows the calling function to perform some logical action by pattern matching for a Nothing return value and reacting accordingly rather than have a program crash when there is no logical return value.

2.1.6 Domain Specific Languages in Haskell

A domain specific language is a computer language which is tailored to solve problems in a specific domain. Domain specific languages can be either external or embedded. An external domain specific language is fully independent of other programming languages meaning that it will have its own compiler. An embedded domain specific language is implemented inside of another programming language and will still use the syntax and compiler of the host language. Embedded domain

specific languages can be classified as either being a shallow embedding or deep embedding (2). In a language which uses shallow embedding there is a direct correlation between what has been written in the host language and what it translates to in the output language. In a deeply embedded domain specific language the host language can perform actions on the code before it is compiled. For example, you may want some evaluations to take place in the host language before outputting to the compiled language.

2.2 Ethereum, Smart Contracts and Solidity

2.2.1 Ethereum

Ethereum is an open source decentralised blockchain released in 2015 which introduced a new cryptocurrency called Ether (3). Ether can be divided into its sub-currencies, the lowest form of which is the Wei which has a value of one quintillionth of an Ether. Ethereum's breakthrough feature was its support for complex smart contract functionality. Previous blockchains such as Bitcoin had only offered very basic smart contract support. The Ethereum runtime environment on which smart contracts are run is called the Ethereum Virtual Machine.

In order to conduct transactions on the Ethereum Virtual Machine there is an associated price referred to as Gas (4). This serves as a way to penalise users for writing computationally expensive smart contracts. Ethereum uses a Proof of work consensus protocol to establish consensus in the blockchain (5). When mining in order to add new blocks to the chain, the 'prover' proves that a certain amount of work has been done by supplying a value that required a large amount of computation to calculate. This value can then be verified easily by the 'verifier'. Ethereum is currently in the process of moving to a proof of stake model with future upgrades of Ethereum, namely Ethereum 2.0 (6). In the envisioned future system users will have to put up their own funds as the stake in order to become a 'validator' who can choose the new blocks to be created. Validators can be punished by other users if they attempt to validate false blocks due to a risk of losing the funds they put up as their stake if they act maliciously.

2.2.2 Smart Contracts

A smart contract is a self-executing contract which is intended to emulate the functionality of a binding agreement between parties. Smart contracts can establish trust between two parties because the contracts are immutable once deployed to the blockchain. The fundamental flaw with smart contracts are the risks that the smart

contract contains an unintended bug which allows some exploit to take place. It is also possible for there to be deliberate loopholes in the code that one of the parties has maliciously inserted in the hopes of exploiting other parties. Additionally, many of the programming languages created for writing smart contracts are relatively new languages and could contain bugs in the way they compile to machine code which is a cause for scepticism when depositing large amounts of funds to smart contracts.

2.2.3 Solidity Programming Language and DAO Hack

Solidity is a high-level programming language created specifically for the writing of smart contracts (7). Once compiled using the Solidity Compiler, Solidity contracts can be run on the Ethereum Virtual Machine where the contracts can then be interacted with by users. The security of Solidity and smart contracts in general has come under increased scrutiny since the DAO hack resulted in the theft of Ether worth approximately $60 million (8). A problem with the DAO's smart contract code allowed for a recursive call exploit by hackers. In order to save the valuation of Ether, a hard fork of the Ethereum blockchain was performed. The original blockchain where the hack occurred is now referred to as Ethereum Classic (ETC) whereas the new blockchain where most of the value has remained is referred to simply as Ethereum (ETH).

Solidity uses function modifiers to distinguish between functions that can and cannot be called with funds. A function declared as payable can be called with funds. By default, functions with no function modifier are non-payable and therefore cannot receive funds. When a payable function is called, the amount of Wei it has been called with will be contained in a variable called 'msg.value'. The address of the account that has called the function can be accessed in Solidity by using the variable 'msg.sender'.

```
1 mapping (address => uint) balance;
2
3 function deposit() public payable {
4     balance[msg.sender] += msg.value;
5 }
```

Listing 2.4: A function to deposit funds to your balance in Solidity.

In listing 2.4 above we see a common deposit function written in Solidity. A variable called balance has been declared in the contract which assigns an unsigned integer amount for each address. The deposit function is declared as public and payable

meaning that messages can be sent to it and these messages can contain funds. When a user calls this function, msg.sender will equal their account address and msg.value will equal the amount of funds they have called the function with. The balance value for their address is incremented by the amount that they have sent so that the contract is able to keep track of how much each user has contributed to the contract balance.

Solidity currently lacks support for floating point numbers. Floating point numbers can be declared but not assigned to in Solidity. To mitigate the effects of this, it is best to use smaller denominations of Ether when doing divisions which may result in a remainder in order to ensure the minimum possible amount is lost to truncation. Solidity supports Time representation with keywords for seconds, hours, days, weeks, months, and years. The current time relative to the Unix Epoch is accessed as 'block.timestamp'. The current time can be used as a seed for the generation of pseudorandom numbers.

2.3 Financial Combinators

Composing Contracts: An Adventure in Financial Engineering is a paper which describes a combinator library for writing financial contracts (9). The paper is a functional pearl which is a paper that sets out an elegant solution to a problem using a functional programming language. The paper demonstrates how simple combinators can be grouped together to describe financial contracts in a clear and concise manner. Logical building blocks like 'zero' representing a contract of value zero and 'give' to transfer contract ownership are implemented in the combinator language to build up a representation of contracts useful in the real-world. The paper shows how real world observables such as times and rates can be modelled into a contract and also describes how to build a value process to assign values to contracts. The paper focuses heavily not just on the ability to represent terms of contracts but also perform high level changes to contracts such as the ability to combine contracts together, allow for changes in contract ownership and scale contracts by some multiple. A huge advantage of this combinator approach is its human readable design and simple syntax.

```
t1 :: Date
t1 = date "1530GMT 1 Jan 2010"

c1 :: Contract
c1 = zcb t1 100 GBP
```

Listing 2.5: Code Snippets used in Composing Contracts Paper (9).

In listing 2.5 we see partial code of a sample contract implemented in the combinators paper. A zero-coupon discount bond (zcb) is represented in a single line with the combinator zcb followed by a date, an amount, and the currency that the amount is in. The ability to understand the main functionality of a contract by reading a single line of code, in this case the assignment to variable c1, ensures that all parties to a contract have a fundamental understanding of its core functionality.

2.4 Cardano and Smart Contracts DSLs

2.4.1 Cardano and Marlowe

Cardano is an open source blockchain platform which uses a proof of stake consensus protocol (10). The native cryptocurrency of Cardano is called Ada. Cardano is focused on ensuring the security and correctness of their platform above all else. The platform is implemented in Haskell to allow for a high degree of formal verification. Cardano is building up a smart contracts language in an attempt to offer the most powerful smart contract feature set of any platform to date.

Marlowe is a domain specific language for generating smart contracts written in Haskell. The language is currently still under construction but aims to generate contracts deployable to the Cardano platform (11). For the modelling of contracts, Marlowe uses Haskell data types to represent the various constructors the language can make use of.

```
data Contract = Close
              | Pay Party Payee Token Value Contract
              | If Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract
              | Assert Observation Contract
```

Listing 2.6: Marlowe's top level Contract data type in Haskell (12).

Some overlaps to the financial combinators paper can be seen such as a data type for

Observables. Other high-level language components include constructors to end a contract, a data type for financial functionality called 'Pay', constructors for if and when conditions, constructors for assertions and constructors to change variable values.

2.4.2 Contract Modelling Language

Contract modelling language (CML) is a domain specific language created in Xtext that allows users to define contracts which will compile to Solidity code (13). The language distinguishes between two basic high-level components: state variables and functions that act on these variables. The language is defined in terms of contract constructs such as the parties of a contract, the assets of a contract and the clause constraints of the contract. These CML constructs have a corresponding Solidity construct which they compile to. For example, a CML asset construct will always compile as a Solidity struct. Contract clauses are defined in terms of a trigger, conditions, actors, modality, actions, and objects. The language uses a separate parser which will compile a .cml file into a Solidity file.

CML deals with Solidity's lack of support for assignment to floating-point numbers by keeping track of a floating point number's value in a normal integer variable without the decimal point but by also keeping track of a scaling factor in a separate integer. For example, the floating point number 2.345 would be represented as an integer of value 2345 and an associated scaling factor of 1000 meaning that its true value is 2345 divided by 1000.

2.4.3 ADICO-Solidity

ADICO-Solidity is a domain specific language built for writing smart contracts that will compile to Solidity code (14). The domain specific language is written in Scala which supports functional programming. The name ADICO comes from the five contract components the DSL uses to represent contracts. Contracts are broken up into attributes (A), deontic (D), aims (I), conditions (C) and or-else statements (O).

In table 2.1 we see how a contract for students to promise to do their homework corresponds to ADICO's components. Similarly to CML constructs, each of the five ADICO components has a predetermined Solidity construct it will compile to. Attributes will always compile as Solidity structs so in this case a struct representing students will be generated. Deontics and Conditions will compile to function modifiers, aims will compile to functions and Or else constructs will compile to flow control statements. At the moment the Solidity code produced by this DSL is

incomplete and only outlines the contract structure with further user work in the Solidity file produced required to complete the contract.

Component	**Example**
Attribute	Students
Deontic	must
Aim	do their homework
Condition	each night
Or else	or else
Attribute	Students
Deontic	may
Aim	receive detention

Table 2.1: Example ADICO Components for Homework Contract.

3 Design

3.1 Requirements Gathering and Design Goals

To ensure that the DSL could generate smart contracts deployable on the Ethereum Virtual Machine it was decided that the DSL should output Solidity files. To determine the requirements for a language that is capable of generating useful Solidity smart contracts, the structure of many commonly written Solidity contracts was studied. It was determined that the DSL functionality should be broken down into financial functionality such as withdraw, deposit, transfer functions and core functionality such as the getting and setting of variables. This was done because financial functionality in Solidity requires special function modifiers such as the payable modifier which allows a function to receive funds. The language would need to allow funds to be sent to the contract to maintain an overall contract balance in certain cases such as games where there is one large pool of money or be able to handle individual account balances for each user of the contract such as when writing a banking contract.

The domain specific language must be able to represent contract variables as well as contract functions. Conditions for functions to be executed on was also factored into the design. Often a time requirement is set on a condition e.g. bids can only be placed between some start and end point. Other conditions factored into the design include variable conditions e.g. a variable being true or one variable being greater than another and conditions on who can call a function e.g. allowing only the contract owner to be able to make a function call. Often times when defining financial contracts an amount of funds being considered will be proportional to some rate e.g. dividends may be paid to shareholders of a company in accordance with their ownership of the company. A requirement set for the domain specific language was to provide a way to make variables proportional to some rates set. The language should provide commonly used variables in Solidity contracts such as balances as data type constructors to make contracts easy to write but should also accommodate users in creating their own unique instances of integers and booleans to allow more complex contracts to be written. The language required a looping structure that could be put over functions to have statements in them executed a number of times dependent on a variable. There needed to be an ability to join financial functions with core functions to produce more complicated functions that had both financial aspects and the ability to update variables. Finally, the contracts outputted by the DSL

needed to be as efficient as possible to ensure a low price of gas (the fee associated with doing work on the EVM) for the user.

Non-functional requirements included a language syntax that was somewhat human readable. It was also important for the language to be succinct to provide a good reason for its use over writing Solidity code in its usual syntax. Additionally, it was important that the domain specific language created was scalable and easy to add more functionality to as more ideas were considered, and a deeper understanding of the Solidity language was achieved throughout the implementation phase.

3.2 Design Phases

In the initial design attempt an abstract syntax was implemented representing the domain specific language that did not compile to an internal representation of the Solidity language grammar. This made it extremely difficult to add new functionality to the language as it continuously further complicated the compilation stages. It also meant that the code written would be less reusable as the compilation step was very specific to the DSL implementation and there was no base level assembly language that represented Solidity in Haskell. Eventually it was decided to implement the entire Solidity language grammar in Solidity and printing functions to get a string representation of the language grammar components so that it would be easier to build domain specific languages on top of this structure and the code would be more useful for future attempts at implementing Solidity DSLs in Haskell. Once the language grammar and printing was implemented it was decided that the language grammar was quite verbose and taking a long time to write components in. Therefore, time was spent designing functions to more easily generate language grammar representations. These functions were written in the Abstractions module. Once the language grammar, language grammar printing and abstractions were implemented, a design for a domain specific languages could be easily implemented as there was a quick way to generate compiled Solidity code simply by converting the domain specific language representation to the Solidity language grammar representation. This allowed for rapid development of a DSL and the ability to quickly roll out sample data constructors to decide on a final design that encapsulated all of the requirements decided on. The design of the DSL could be easily tweaked when implementing sample contracts if any functionality was missing. Therefore the design of the lower-level DSL components was constantly changing throughout the implementation phase but the high-level design generally stayed the same.

3.3 Final Language Structure

3.3.1 High-Level Language Components

The top-level data type of the DSL is simply a data type called Contract. Although Solidity allows contracts to create instances of other contracts, for the purpose of writing financial contracts and keeping in mind the goal to keep the language as simple syntactically as possible, a more complex top level is not necessary. A Contract will have a name associated with it and a list of contract elements which will break down into just two categories of declarations, setters or function elements corresponding to Solidity contract-level variables and Solidity functions respectively. Set takes a variable to set as a contract level variable. When the constructor for the contract is being generated during the compilation phase, parameters to the constructor will need to be automatically created to assign the setters initial values on contract creation where appropriate. This avoids the need for users to define their own constructors. Function elements take a string representing the name of the function and a function data type. This function data type can be either a core or financial function or one of the recursive function data types which are ProportionalTo, Loop, Conditioned and Join. Each of these function types will then also take lower level language components to further specify their functionality. This high-level structure of the language allows for easy expansion of the DSL. When new functionality is required it can be implemented as an extra pattern match in one of the lower level data types. For example, adding an extra condition involves simply adding a new pattern match for conditions and adding a new rate involves simply adding a new pattern match acting on ProportionalTo. This very simple high-level language structure comes at the cost of some features for the user such as not being able to pass lower-level compile details such as the compiler version or file imports to a contract.

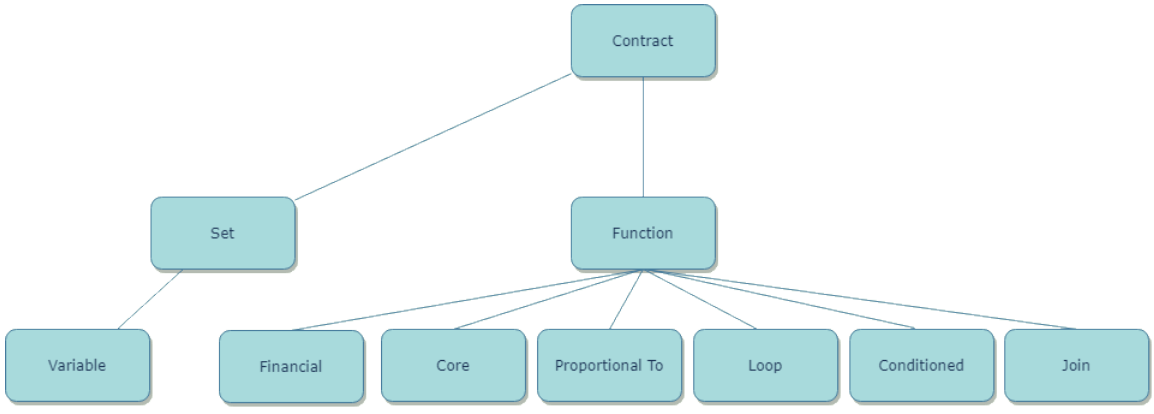

Figure 3.1: High-level view of language structure.

3.3.2 Low-Level Language Components

Figure 3.2 shows the lower level language structures which the high-level structures use. For example a function that has been specified as financial will make use of an instance of financial functionality to specify the action being performed as well as an amount to specify the amount of funds in question and a source to specify what the source of the funds in question is.

Condition	Financial Functionality	Variable		TimeRequirement	CoreFunctionality
RequireOwner	Withdraw	Owner	Interst	Started	GiveOwnership
RequireTime	Deposit	ContractBal	Tax	NotStarted	GetBalance
RequireRecipient	Transfer	StartTime	DIRT	Ended	GetVariable
RequireNotZero	BlankPayable	EndTime	SharesOwned	NotEnded	BecomeRecipient
RequireTrue	Source	RecipientAddress	TotalShares	BetweenStartAndEnd	Update
RequireFalse	ContractBalance	Balances	MessageValue	VariableRelation	
RequireVariableRelation	UserBalance	Addresses	BoolTrue	ValueGreaterThan	
Amount	Rate	MessageSender	BoolFalse	ValueLessThan	
All	InterestRate	IndexAddresses	Random	ValueGreaterThanEqualTo	
SpecificAmount	TaxRate	SignedAmount	AddressList	ValueLessThanEqualTo	
	Shares	UnsignedAmount	IndexAddressList	ValueEqualTo	
	InterestRateLessDIRT	BooleanVariable	Increment	ValueNotEqualTo	
		Decrement			

Figure 3.2: Lower level language combinators.

4 Implementation

4.1 Overview of Implementation

There were four stages involved in the implementation of the DSL. The first stage involved fully implementing a representation of the Solidity language grammar into Haskell. This involved implementing 96 data types in Haskell that corresponded with the 96 language rules specified in the Solidity documentation. This stage was crucial as it meant that if the DSL representation of a contract could be translated to the Solidity language grammar representation of a contract through conversion functions then the ability to compile the contract to Solidity code would be guaranteed. In effect, it provided a kind of assembly language for the DSL to assemble to from which it could then be outputted as Solidity syntax. The second stage involved supplying printing functions for all 96 language rules of Solidity. These printing functions could take the Haskell data types representing the Solidity language grammar and convert them to strings representing valid Solidity syntax. A contract represented in the Solidity language grammar could be passed to the print function for that rule which would output a string that could then simply be written to a Solidity file to provide program output. The third stage of implementing the DSL involved creating functions to write the Haskell language grammar representation of Solidity more quickly and easily. This stage was necessary as the language grammar representation is quite verbose and consequently very time consuming to write. This would save time when translating the DSL data types into the Haskell language grammar representation of Solidity and would also make the code more reusable for future attempts to implement Haskell eDSLs that compile to Solidity. The final stage of implementation involved representing the DSL as data types in Haskell and providing conversion functions from these DSL data types into the Solidity language grammar representation data types to allow for compilation to Solidity files. Useful example contracts were then written in the DSL and compiled to Solidity to demonstrate the implementation.

Figure 4.1: Compilation from DSL implementation of a contract to Solidity syntax.

4.2 Solidity Language Grammar

The first stage of implementing the domain specific language was to translate the Solidity Language Grammar into Haskell. This was achieved by individually creating a new data type in Haskell for each language grammar rule provided in the Solidity documentation. The Solidity documentation specifies the entire language grammar from the top level down. In total, Solidity uses 96 language grammar rules for specifying the structure of the language in version 0.8.1. This includes the language rules for the Yul language grammar which is a lower level language for writing smart contract that can also be compiled using the Solidity compiler. All 96 language grammar rules were translated into Haskell as new data types. Each data type could have multiple branches corresponding to the multiple paths that the rule could be followed down. Figure 4.2 shows the top-level Solidity language grammar rule specified in the Solidity documentation.

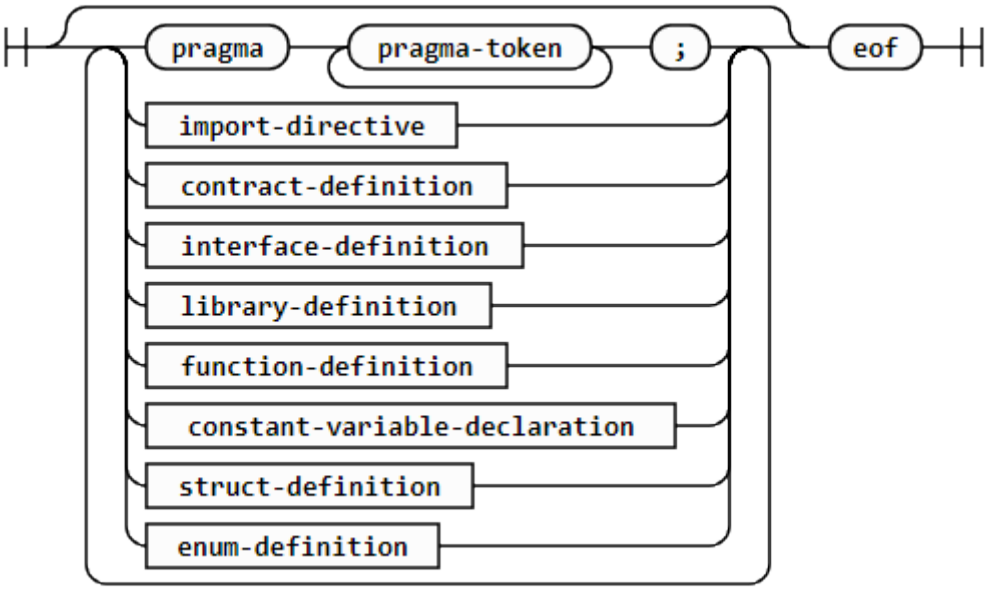

Figure 4.2: Top level rule of the Solidity Language Grammar (15).

Each of the rectangular constructs then leads to another language grammar rule. For example, following the function-definition rule of the language grammar will lead to the rule seen in figure 4.3.

```haskell
data Solidity = EOF
              | Pragma PragmaToken [PragmaToken] Solidity
              | ImportDir ImportDirective Solidity
              | ContractDef ContractDefinition Solidity
              | InterfaceDef InterfaceDefinition Solidity
              | LibraryDef LibraryDefinition Solidity
              | FunctionDef FunctionDefinition Solidity
              | ConstVariableDec ConstantVariableDeclaration Solidity
              | StructDef StructDefinition Solidity
              | EnumDef EnumDefinition Solidity
```

Listing 4.1: Top level Language Grammar rule implemented in Haskell.

In listing 4.1 we see the Haskell implementation for the top-level Solidity language rule. A name must be provided for each constructor e.g. Pragma and then the arguments the constructor takes is specified. Haskell's pattern matching structure provides a clear translation from the language grammar rules to the Haskell representation. EOF takes no arguments to construct a contract as it represents the empty contract. Each of the pattern matches for the top-level rule is recursive as they can take another Solidity data type as an argument except for EOF which does not have any arguments.

The pragma constructor in 4.2 specifies that one or more pragma-tokens must be supplied when using this constructor. The corresponding Haskell implementation for this pattern in 4.1 specifies Pragma PragmaToken [PragmaToken] Solidity. This means that supplying at least one PragmaToken is mandatory when using this constructor. If supplying only one PragmaToken then the list can be left empty. If supplying more than one PragmaToken then the first PragmaToken will be placed outside of the list and the remainder inside of the list. Had the first PragmaToken before the list not been specified then it would have been possible to supply an empty list when constructing this rule which is not what the documentation specifies.

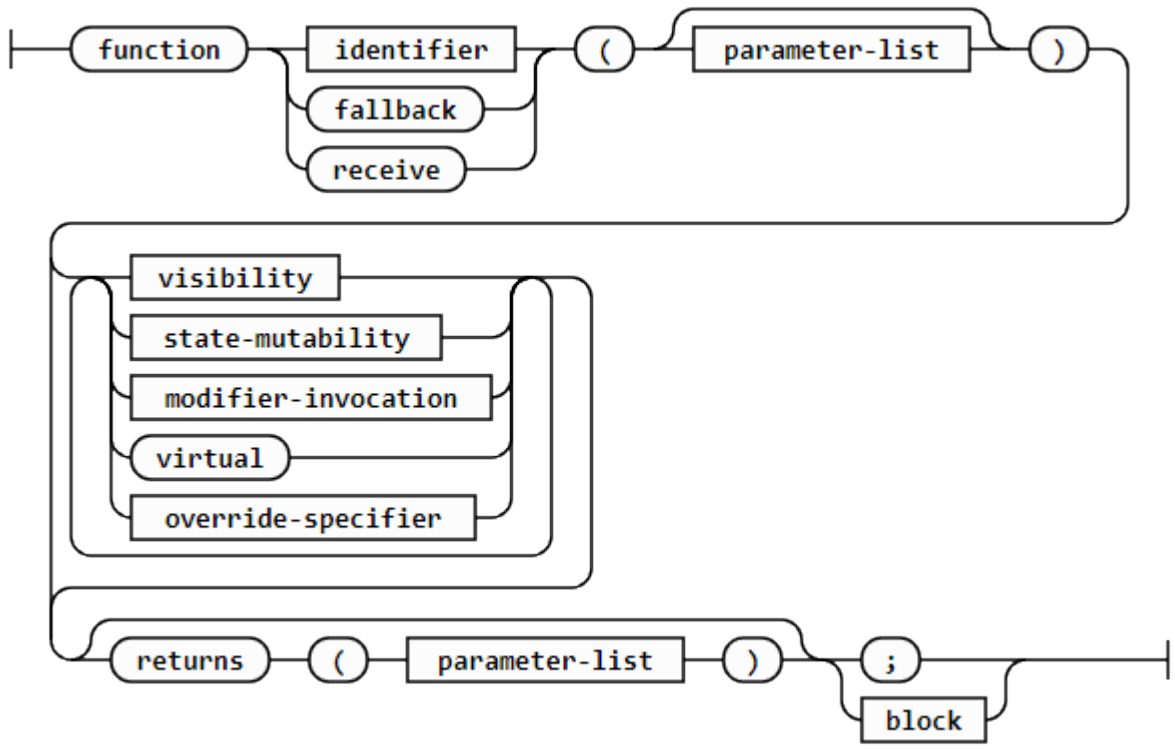

Figure 4.3: Function-definition rule of the Solidity Language Grammar (15).

```
1 data FunctionDefinition = FunctionDefinition FunctionName
2   (Maybe ParameterList) [FunctionModifiers]
3   (Maybe ParameterList) (Maybe Block)
```

Listing 4.2: Function-definition Language Grammar rule implemented in Haskell.

The FunctionDef constructor seen in listing 4.1 specifies that a FunctionDefinition and Solidity data type must be supplied. In listing 4.2 we can see the Haskell implementation of the FunctionDefinition rule.

The three ways to specify the function name as either an identifier, fallback function or receive function has been further abstracted out to a data type called FunctionName. Similarly, the various modifiers the function can have are abstracted out to a data type called FunctionModifiers. For some complex rules with many branches such as FunctionDefinition, additional data types were implemented for code clarity. Optional components of language grammar rules such as the optional parameter-lists in the FunctionDefinition rule are implemented using Haskell's Maybe data type. This allows instantiations as either Nothing if a ParameterList is not being supplied or Just followed by a ParameterList if one is being supplied.

```
coinExample = (Pragma (PragmaToken 's') [(PragmaToken 'o'),
 (PragmaToken 'l'), (PragmaToken 'i'), (PragmaToken 'd'),
 (PragmaToken 'i'), (PragmaToken 't'), (PragmaToken 'y'),
 (PragmaToken ' '), (PragmaToken '^'), (PragmaToken '0'),
 (PragmaToken '.'), (PragmaToken '7'), (PragmaToken '.'),
 (PragmaToken '4')] contractDef)
```

Listing 4.3: Example of a top level language grammar rule instantiation in Haskell.

In listing 4.3, an instansiation of the top-level language grammar rule is shown. It specifies the version of Solidity the contract should be compiled in and then passes in a variable called contractDef supplying the rest of the contract defined in terms of the language grammar elsewhere. It is clear to see just from the version specification that the language grammar implementation is quite verbose to write. For instance, each PragmaToken can only represent a single character, so it takes 15 of them to specify the compiler version. The abstractions section of the implementation shows how simple functions were implemented which can quickly generate the same contract with far less user input in order to make it far easier to write instantiations of the language rules in Haskell and to speed up the process of converting the top level DSL that will be implemented into its language grammar representation for compilation.

4.3 Language Grammar Printing

To ensure that the full Haskell representation of the Solidity language grammar could be compiled to valid Solidity code, a print function was implemented for each of the 96 language grammar rules. Each print function produces a string as specified by the syntax format in the Solidity language grammar documentation.

```
printSolidity :: Solidity -> String
printSolidity (EOF) = ""
printSolidity (Pragma token tokenList nextHighLevel) = "pragma "
  ++ (printPragmaToken token)
  ++ (printPragmaTokenList tokenList) ++ ";\n"
  ++ (printSolidity nextHighLevel)
printSolidity (ImportDir directive nextHighLevel) =
  (printImportDirective directive) ++
  (printSolidity nextHighLevel)
printSolidity (ContractDef definition nextHighLevel) =
  (printContractDefinition definition) ++
  (printSolidity nextHighLevel)
printSolidity (InterfaceDef definition nextHighLevel) =
  (printInterfaceDefinition definition) ++
  (printSolidity nextHighLevel)
printSolidity (LibraryDef definition nextHighLevel) =
  (printLibraryDefinition definition) ++
  (printSolidity nextHighLevel)
printSolidity (FunctionDef definition nextHighLevel) =
  (printFunctionDefinition definition 0) ++
  (printSolidity nextHighLevel)
printSolidity (ConstVariableDec declaration nextHighLevel) =
  (printConstVariableDeclaration declaration) ++
  (printSolidity nextHighLevel)
printSolidity (StructDef struct nextHighLevel) =
  (printStructDefinition struct 0) ++
  (printSolidity nextHighLevel)
printSolidity (EnumDef definition nextHighLevel) =
  (printEnumDefinition definition 0) ++
  (printSolidity nextHighLevel)
```

Listing 4.4: Top-level print function to convert language grammar contract to string.

Listing 4.4 shows the top-level printing function for the language grammar. EOF specifies the end of file and so will produce an empty String. All other top-level constructors that Solidity can instantiate to are calling lower level print functions based on their pattern match. As all these top-level constructors, excluding EOF, can contain further Solidity data types, there is then a recursive call to printSolidity with the remaining Solidity data type they contain.

One problem with the language grammar documentation is that no formatting for spaces or tabs is specified for Solidity. This means that following the language grammar directly would output a Solidity file just one line long containing all the code. While this would still compile correctly, it would not be easily human readable.

In order to output a cleanly formatted file, an integer called tabCount is passed to most functions that print out blocks of statements. This allows the printing to keep track of how many tabs should be printed before a rule. For example, when printing a for statement block inside of a function, printForStatement will be called with tabCount incremented by one to indent its statements.

```
printWhileStatement :: WhileStatement -> Int -> String
printWhileStatement (WhileStatement expression statement) tabCount =
    "while (" ++ (printExpression expression) ++ ") "
    ++ (printStatement statement (tabCount + 1))
```

Listing 4.5: Function to print while statements showing tab count formatting.

Listing 4.5 shows a function for printing while statements. Before the function is called, the tabs needed to be printed before the 'while' keyword are already printed by the calling function. The printWhileStatement can then start by printing the 'while' keyword as well as the brackets around the expression being evaluated. A lower level print function is called to print the expression inside the brackets. When the while function is calling the printStatement function to print out the contents of its loop it passes in an incremented tabCount variable so that the contents of the loop are outputted with a tab indentation inside the while statement.

When a data type contains a Maybe type inside of it to specify optionality, the corresponding print function will have to be able to pattern match for both the Nothing and Just instantiation to ensure it can produce the correct string in both scenarios. For each Maybe type in a data constructor the number of pattern matches in the corresponding print function will double. For example, the modifier definition data type contains three Maybe types. This means that the print function for a modifier definition has eight (2^3) pattern matches to capture all of the possible combinations of the three Maybe type instantiations.

```
outputSolidityContract :: Solidity -> String -> IO ()
outputSolidityContract contract destination = do
  writeFile destination ""
  appendFile destination "// SPDX-License-Identifier: GPL-3.0\n"
  appendFile destination (printSolidity contract)
```

Listing 4.6: Function to output language grammar representation of a contract to a file.

To create a function that can take the language grammar representation of a contract and produce a Solidity file it then only needs to call the printSolidity function with that contract to obtain the Solidity syntax. The outputSolidityContract function will

take the contract represented in the language grammar implementation and a string representing the output destination. It will first clear the file by writing an empty string to it to ensure the file is empty. It will then add an SPDX-License-Identifier which the Solidity language requires at the top of Solidity files, but this is not specified in the language grammar followed by appending the string produced by the printSolidity function acting on the contract.

4.4 Abstractions

After having achieved an implementation of the Solidity language grammar in Haskell and having a way of compiling contracts represented in the grammar to Solidity files, functions were then created which would allow the language grammar to be written in a less verbose manner. The many layers of rules within rules in the Solidity language grammar made it difficult to implement instances of the language grammar without constantly checking the data types and this would have made the final step of implementing the DSL that converts to the language grammar much more time consuming. To resolve this, time was spent writing functions to more easily create Solidity types, variables, functions, expressions, and statements represented in the Solidity language grammar data types. A total of 72 such functions were defined that would return language grammar data types from a handful of parameters saving significant time in the DSL implementation step. Included in these were helper functions that could perform useful actions on the language grammar data types which would be needed during the DSL implementation step such as the ability to join two functions together.

Type Abstractions

In order to specify types more quickly than having to provide both the type-name and type keyword, Haskell variables were used to represent the types as single keywords. This was done for all integer types from size 8 bytes to size 256 bytes as well as for unsigned integers, all bytes sizes, addresses, booleans and strings. Listing 4.7 shows some of the type abstractions for integer types and this was continued in increments of 8 bytes up to the maximum Solidity size for integers.

```
int = SignedIntType Int
int8 = SignedIntType Int8
int16 = SignedIntType Int16
int24 = SignedIntType Int24
int32 = SignedIntType Int32
int40 = SignedIntType Int40
int48 = SignedIntType Int48
int56 = SignedIntType Int56
int64 = SignedIntType Int64
```

Listing 4.7: Partial abstractions for integer types.

Variable Abstractions

Variable declarations in the language grammar are quite common but similarly quite verbose to write. There are four main categories of declarations for each variable type: state variable declarations, state variable assignment declarations, non-state variable declarations, and non-state variable assignment declarations. Functions to write all four were written for the main types such as addresses, booleans, strings, bytes, integers, and unsigned integers. More complex functions were implemented for struct declarations and mapping declarations.

```
intVariableAssignmentDeclaration :: String -> Expression
  -> VariableDeclarationStatement
intVariableAssignmentDeclaration name e =
  SingleVariableDeclartion (VariableDeclaration
  (ElementaryType $ SignedIntType Int) (Nothing)
  (createIdentifier name)) (Just e)
```

Listing 4.8: Function to generate language grammar integer assignment declaration.

In listing 4.8 we see a function to quickly declare and assign to an integer by providing a string representing the variable's name and an expression representing its value.

Functions Abstractions

The main function abstractions needed were the ability to write void and non-void functions by passing in a few parameters. Listing 4.9 shows a function for quickly generating non-void functions in the appropriate language grammar data type by supplying the function name (represented as a string), function modifiers, an input parameter list, a block containing the function statements, and the function return

type. The function has two pattern matches corresponding to whether or not the function input parameter list is Nothing (representing a function that takes no parameters) or a parameter list wrapped in a Just (representing a function that does take parameters). The function for generating void functions is similar but without the second parameter list argument as there is no return parameter being specified.

```
createReturnFunction :: String -> [FunctionModifiers]
  -> Maybe ParameterList -> Block -> ParameterList
  -> FunctionDefinition
createReturnFunction name modifiers
(Just parameters) block returnParams =
  FunctionDefinition (IdentifierName $
  createIdentifier name) (Just parameters)
  modifiers (Just returnParams) (Just block)
createReturnFunction name modifiers (Nothing)
block returnParams =
  FunctionDefinition (IdentifierName $
  createIdentifier name) (Nothing)
  modifiers (Just returnParams) (Just block)
```

Listing 4.9: A Function to quickly generate a non-void language grammar function.

General Abstractions

In listing 4.3 it was shown how verbose writing many of the data types such as pragma tokens could be. For this reason, many of the particularly verbose and commonly used language grammar data types for low level features were noted and abstraction functions implemented. These included identifiers which are used for specifying function and variable names. Where previously a list of single characters for each character in the name would have to be supplied, a function that could generate an identifier from a normal Haskell string was introduced to speed this process up dramatically.

```
shorternedCoinExample = createPragma "solidity ^0.7.4" contractDef
```

Listing 4.10: Using an abstraction function to quickly specify the Solidity version.

Listing 4.10 shows the use of one of the general abstraction functions which allows a single string to be used to specify a contract's Solidity version rather than a long list of single character pragma tokens.

Helper Functions

Looking ahead to the DSL implementation, it was clear that functions that could perform high-level manipulations of the data types would be needed. This included the ability to join functions together and change function modifiers. Other helper functions such as functions to check whether a function has the payable modifier and functions to join two blocks of statements would also become very useful. Listing 4.11 shows a function to join two parameter lists (both wrapped in the Maybe type) together which is useful when joining two functions together. If both parameter lists are Nothing (representing the empty parameter list) then the result is Nothing. If only one parameter list is not a Nothing then return that list. If both parameter lists contain parameters, then another helper function is called to join their contents together and the result is wrapped in a Just and returned.

```
1 joinParamLists :: (Maybe ParameterList) -> (Maybe ParameterList)
2     -> (Maybe ParameterList)
3 joinParamLists (Nothing) (Nothing) = Nothing
4 joinParamLists (Just x) (Nothing) = Just x
5 joinParamLists (Nothing) (Just y) = Just y
6 joinParamLists (Just x) (Just y) = Just (joinParams x y)
```

Listing 4.11: Function to join two parameter lists quickly.

By providing an implementation of the Solidity language grammar in Haskell, printing functions to turn the language grammar into valid Solidity syntax and functions to quickly generate the language grammar, it is then quick to write a DSL with data types that can assemble to the language grammar data types and then be turned into valid Solidity syntax. This implementation method also makes the code easily reusable for any other attempts at writing a domain specific language in Haskell for generating Solidity code. A programmer would simply need to map out their DSL abstract syntax and then specify how this abstract syntax gets translated to the Solidity language grammar representation.

4.5 DSL Implementation

The domain specific language created is represented as Haskell data types. There are data types for each of the main Contract elements and then pattern matches on those data types for each of its instantiations. For example, Variable is a data type and an unsigned integer amount is a pattern match of that data type. For each of these data types and their various pattern matches there are conversion functions with the ability to take the type in the DSL representation and convert it to the Solidity

language grammar representation from which it can be outputted to Solidity syntax.

4.5.1 Contract

The highest-level data type of the DSL is the Contract data type. This data type takes a string and a list of ContractElement data types (explained in 4.5.2). The string represents the name of the contract. The contract elements represent the contract's state variables and functions. This top-level Contract data type gets converted to a data Solidity which is the top level rule of the Solidity language grammar from which it can be outputted as Solidity syntax.

```
1 data Contract = Contract String [ContractElement]
2
3 data ContractElement = Set Variable | FunctionElement String Function
```

Listing 4.12: The two highest-level data types of the DSL.

4.5.2 Contract Element

Set

Set is a pattern match for the ContractElement data type. Set takes one argument, a variable data type (4.5.9). Set represents a state variable that the contract needs. For example, Set ContractBal will set a public unsigned integer called contractBalance representing the entire balance of funds that the contract possesses. When a set variable is being converted to Solidity it corresponds to a state variable element in the Solidity language grammar. Set variables undergo an extra step in their conversion to Solidity when they are passed to the buildConstructor function. This function will go through all contract variables being specified for the contract and decide how to build a constructor containing all the necessary components. For example, if you specify that you are setting a start time and an end time then the constructor will set start equal to the time that the contract was created and will pass in a parameter to the constructor called secondsAfter which is the number of seconds from start to end specified as an unsigned integer. The end time will then be set to the start time plus the number of seconds specified. The constructor has some special cases where it decides a value to be set. For example, if you specify that you are setting the contract balance variable then the constructor will automatically set this balance to zero on contract creation as this is the balance expected before any funds have been sent to the contract.

Function Element

The FunctionElement pattern match for the ContractElement data type encapsulates all of the ways functions can be built with the DSL. It takes a string representing the name of the function being built as well as a Function data type representing its functionality. The Function data type can be either financial (4.5.3), core (4.5.4), proportional to (4.5.5), loop (4.5.6), conditioned (4.5.7) or join (4.5.8).

4.5.3 Financial Functionality

The Financial constructor takes three arguments: a data type called FinancialFunctionality specifying what financial action is being performed, a data type called Amount representing how much funds is being talked about and a data type called Source representing the location of the funds in question. The FinancialFunctionality data type has four constructors: Withdraw, Deposit, Transfer and BlankPayable. Withdraw will add core functionality to the function to allow some specified amount of funds to be withdrawn from some source specified. Deposit will allow some amount of funds specified to be deposited to a source specified. Similarly, Transfer will facilitate the transfer of a specified amount of funds from a specified source. BlankPayable is a special instance of the FinancialFunctionality data type as it ignores the source and amount provided. Its only purpose is to provide a function with the "Payable" modifier which will allow the function to be called with an Ether amount. This is useful when the user needs a function with the payable modifier but does not need deposit, withdraw or transfer functionality such as when they want to join a CoreFunctionality (4.5.4) function with a BlankPayable function to allow a function to receive funds. In table 4.1 we can see how the combination of FinancialFunctionality, Amount and Source will combine to create useful financial functions.

Functionality	Amount	Source	Explanation
Withdraw	All	ContractBalance	Withdraws everything from contract balance
Deposit	All	ContractBalance	Deposits msg.value to contract balance
Transfer	All	ContractBalance	Transfers entire contract balance to a user's balance
Withdraw	SpecificAmount	ContractBalance	Withdraws a specific amount from the contract balance
Deposit	SpecificAmount	ContractBalance	Deposits a specific amount to the contract balance
Transfer	SpecificAmount	ContractBalance	Transfers specific amount from contract value to a user balance
Withdraw	All	UserBalance	Withdraws everything from a user's balance
Deposit	All	UserBalance	Deposits msg.value to a user's balance
Transfer	All	UserBalance	Transfer all from one user's balance to another
Withdraw	SpecificAmount	UserBalance	Withdraws a specific amount from a user's balance
Deposit	SpecificAmount	UserBalance	Deposits a specific amount to a user's balance
Transfer	SpecificAmount	UserBalance	Transfers a specific amount from one user's balance to another's
BlankPayable	<Ignored>	<Ignored>	Provides a blank function with the Payable modifier

Table 4.1: Table showing how financial constructors combine.

4.5.4 Core Functionality

The Core constructor for a function takes a single argument called CoreFunctionality. CoreFunctionality is a data type with five different possible pattern matches. CoreFunctionality can branch to GiveOwnership which will provide a function which takes an address as a parameter and will override the owner of the contract to this new address. GetBalance will provide the functionality for a user to check their balance. GetVariable will provide functionality to get the value of some variable e.g. a lotto contract may have a recipient called winner representing the winner of the lottery. GetVariable could be combined with this winner variable to create a function that returns who the winner is as seen in listing 4.13.

```
1 checkWinner = FunctionElement "checkWinner"
2   (Core (GetVariable (RecipientAddress winner)))
```

Listing 4.13: A function to check who the winner is using Core Functionality.

BecomeRecipient provides the functionality for someone new to become some specified recipient e.g. updating who the winner of a game is. Finally, Update will provide the ability for a variable to be updated to some new value e.g. updating the highest bid in an auction. Each of these five data types that can be supplied to Core has a Haskell function that will use the supplied values to create the function in the language grammar representation of Solidity.

4.5.5 ProportionalTo

ProportionalTo takes two variables, a rate and a function as its arguments. ProportionalTo will add a statement to the start of the function supplied to it which makes the first variable equal to the second variable manipulated based on some rate. The four rates that it works with are interest rates, interest rates less DIRT, tax rates, and share ownership. Table 4.2 shows how the rate specifies the actions ProportionalTo will perform.

InterestRate

This rate is useful when paying interest to people based on their holdings in a bank. It can be used to set some variable equal to a user's balance in the contract multiplied by the interest rate. Due to Solidity's lack of support for floating point numbers, interest rate is represented as an unsigned integer representing the rate percentage. For example, if the interest rate is 10% then the interest rate variable is set to 10. The user's balance is multiplied by 10 and the result is then divided by 100 to get their interest owed.

InterestRateLessDIRT

This rate is similar to interest rate but will subtract the DIRT rate in the contract from the amount of interest being paid. After getting the amount of interest owed to a user it will subtract the percentage of DIRT from that amount owed. For example, if the DIRT rate is 12% then the amount of interest being paid is made 88% of what it was originally.

TaxRate

This rate is used when collecting taxes from someone's balance. It can be used to set an amount equal to someone's balance times the tax rate to see how much is owed to the tax collector. It is most useful when writing a contract that keeps track of a user's profits made so that they are just paying taxes on their profits.

Shares

Shares is useful when paying dividends to people with ownership in a company or specifying voting power based on ownership. For example, it can be used to make the amount being paid to a shareholder proportional to that person's ownership stake. The amount being paid will be multiplied by the recipient's number of shares and the result will be divided by the total number of shares a company is divided into. Listing 4.14 shows a function written in the DSL to transfer money from the

contract into someone's account based on their
shares in the company. It is specified that only the contract owner can call this function.

```
payDividends = FunctionElement "payDividends"
  (Conditioned (RequireOwner) (ProportionalTo (SignedAmount "amount")
  (IndexAddresses (SharesOwned) (Recipient "_to")) (Shares)
  (Financial Transfer SpecificAmount ContractBalance)))
```

Listing 4.14: A function to pay dividends to someone based on their ownership stake.

Functionality	Rate	Explanation
ProportionalTo	InterestRate	Takes two variables and a function and sets the first variable equal to the second multiplied by the interest rate in the function.
ProportionalTo	InterestRateLessDIRT	Takes two variables and a function and sets the first variable equal to the second multiplied by the interest rate less DIRT in the function.
ProportionalTo	TaxRate	Takes two variables and a function and sets the first variable equal to the second multiplied by the tax rate in the function.
ProportionalTo	Shares	Takes two variables and a function and sets the first variable equal to the second divided by the total number of shares in the contract.

Table 4.2: Table showing how ProportionalTo constructors combine.

4.5.6 Loop

The loop constructor can be used to have a function's statements loop a certain number of times. Loop takes a variable data type which should be an integer or unsigned integer specifying the number of times the function should loop for and the function to act on and will wrap the contents of the function's block of statements within a for loop executing that number of times.

4.5.7 Conditioned

The Conditioned constructor will take a Condition and a function and stipulate that the function's statements are only executed given that the condition passes. This allows permissions for functions to be defined such as who can call a function and when. Conditions were converted to the Solidity language grammar as either an if statement over the function's execution statements or as the require keyword in Solidity that exits a function if the requirement specified is not true. The following conditions have been defined in the language:

RequireOwner

This stipulates that only the owner of the contract can call this function successfully.

RequireTime

RequireTime takes a TimeRequirement constructor and specifies that the function can only be called successfully based on the time requirement being true. There are time requirements to check that an event has started, has not started, has ended, has not ended and is somewhere between the start and end point.

RequireRecipient

This will specify that you must be a certain recipient based on the recipient data type passed into this constructor. For example, you may specify that only the recipient known as the winner can withdraw from the contract balance.

RequireNotZero

RequireNotZero takes a variable data type as an argument and adds a requirement check that the variable does not equal to zero. If it does equal zero, then the function will exit immediately.

RequireTrue and RequireFalse

These conditions will check that a supplied variable is true or false in order for the function to proceed.

RequireVariableRelation

RequireVariableRelation takes a VariableRelation data type and adds a requirement for that variable relation. VariableRelation will itself take two variables and can check if one is greater than, less than, greater than or equal to, less than or equal to, equal to or not equal to the other.

4.5.8 Join

Join is used to join two functions together and create a new function which contains the statements of the first function followed by the statements of the second. The parameter lists of the functions will be combined and also their function modifiers will be combined. Join can be used on all instances of the Function data type including being used to join core functions and financial functions and to join other

functions that are the result of join operations themselves. Join takes two arguments both of which are function data types representing the functionality being joined. If the two functions being joined have conflicting function modifiers, such as one being a view only function while the other is a Payable function, join will always keep the most powerful function modifier, in that case the Payable modifier. Join is what allows long complex functions to be built from the simple function properties defined in the FinancialFunctionality and CoreFunctionality data types.

4.5.9 Variable

Variable has constructors for some of the most commonly used variables in Solidity contracts and can also be used to create user defined booleans, addresses, address mappings, address lists, signed integers and unsigned integers. The following are explanations of each of the available variable constructors:

Owner

Setting owner will set an address payable called owner for the contract. It will also mean that the constructor generated for the contract will set the original creator of the contract (‘msg.sender’ in Solidity) as the owner of the contract by default.

ContractBal

Often times the concept of a contract balance for the entire contract is needed rather than an individual balance for each user. This variable will set an unsigned integer called contractBalance and the constructor will initialise it to 0 on contract creation to represent the fact that the contract is starting with zero funds until any funds has been sent to it.

StartTime

This will define an unsigned integer called start and will have the constructor initialise start as the time of contract creation (represented as ’block.timestamp’ in Solidity). Start can be compared against the current time to check if someone should be able to interact with some functionality at the current time.

EndTime

This will define an unsigned integer called end and will have the constructor take a parameter called secondsAfter specifying how long the contract should last in seconds. End is then set to start time plus the number of seconds the contract is running for. Similarly to start, end can be compared against the current time to check

if someone should be able to interact with some functionality at the current time.

RecipientAddress

This will create a new address payable with the name specified by the Recipient passed into this constructor. This is useful for defining people who should be sent funds e.g. a winner or tax collector.

Balances

Setting balances will provide a mapping called balance from each address to an unsigned integer which represents that user's balance. This is useful for keeping track of peoples' ownership of funds such as when writing contracts representing banks.

Addresses

Addresses takes a string as an argument and will create a new mapping variable from addresses to unsigned integers. The name of the variable will be the same as the string passed in to Addresses. It is similar to the use of Balances but can be used to create user defined mappings when balance is not the appropriate name or more mappings are required than just balances.

MessageSender

MesssageSender is a special variable which is a reference to the Solidity value 'msg.sender' representing the address of the person calling a function. Often times this variable needs to be referenced e.g. when somebody is calling a function to deposit funds to their account, the funds should go to the balance of their address. Since this is a special case this variable will have no effect when passed in with Set as it cannot be a contract variable and should only be used within functions.

IndexAddresses

The IndexAddresses constructor takes a variable and a recipient data type as its arguments. It is used to index some variable with a recipient. For example, we may have a recipient called winner and want to find out the balance of this recipient. In that case we would pass in the Balances constructor as the variable and the winner as the recipient into the IndexAddresses constructor.

SignedAmount

SignedAmount takes a string and will create an integer variable with the same name as the supplied string. The constructor will automatically take in an extra parameter to initialise this integer to.

UnSignedAmount

UnSignedAmount takes a string and will create an unsigned integer variable with the same name as the supplied string. The constructor will automatically take in an extra parameter to initialise this unsigned integer to.

BooleanVariable

This constructor takes a string representing the variable's name and creates a boolean with that name, initialising it in the constructor by having the constructor take a boolean parameter representing its initial value.

Interest, Tax, DIRT

These three constructors can be passed to Set and will set unsigned variables representing an interest rate, tax rate and DIRT rate. They will also have variables passed to the constructor to initialise them to values. They are useful when combined with the ProportionalTo constructor to allow financial transactions the ability to factor in laws and regulations.

SharesOwned, TotalShares

SharesOwned will set a mapping from addresses to an unsigned integer representing the number of shares each address owns. TotalShares will set an unsigned integer representing the total number of shares a company is divided into. These constructors can be used with ProportionalTo to make dividends payments proportional to someone's ownership of a company or to give someone voting power based on their percentage ownership.

MessageValue

Similar to MessageSender, MessageValue is a special variable in Solidity. It represents Solidity's in built 'msg.value' which is the amount of funds that a payable function has been called with. Combining Set with MessageValue will have no effect as MessageValue cannot be a contract variable and should only be used within functions.

BoolTrue, BoolFalse

BoolTrue and BoolFalse are special variables representing the true and false keywords in Solidity. Similar to MessageSender and MessageValue they cannot be set as contract variables and trying to do so will have no effect on the outputted contract. They are useful when updating user defined booleans to new values.

Random

Random takes a variable as an argument which should be a signed or unsigned integer and will generate a random number between zero and the passed in variable's value minus 1. This is useful when introducing randomness to a contract. For example, you may have a list of players who have bought in to a lottery. If the list is of size 6 you would want to generate a number between 0 and 5 to get an element in the list to choose as your winner. The randomness constructor will use a combination of the current time (block.timestamp) and the difficulty to create the next block (block.difficulty) in Solidity to generate a pseudo-random integer.

AddressList

AddressList takes a string representing the name of the list of addresses you wish to create. This is useful when a list which can vary in size is needed. For example, you may wish to add a player's address to a list of addresses called players when they buy in to a lottery.

IndexAddressList

IndexAddressList takes two variable, the first being the address list being indexed into and the second being the index being accessed.

Increment, Decrement

Increment and Decrement are two constructors which take a variable and an integer and will increment or decrement that variable by the integer supplied. For example, to increment a signed integer by one the first argument of Increment would be the signed integer and the second argument of Increment would be the number one.

4.5.10 Output to Solidity

The domain specific language is outputted to Solidity by first being translated to the Solidity language grammar representation built in Haskell and from there it can be

printed using the printing methods defined for the language grammar to create a String representation of the contract that can be written to a file. The top-level Contract data type is passed to the outputContract function. This function will create a Solidity language grammar representation of the DSL contract by creating a Solidity contract of the same name as the string supplied in the data type and by converting all of the ContractElements specified in the DSL to their equivalent representation in the Solidity language grammar by passing the list of ContractElements to the convertToSolidity function. The function convertToSolidity will take each ContractElement in this list and convert it to a representation in the language grammar. The top-level contract element in the Solidity language grammar is called a ContractBodyElement. Therefore, a translation from each DSL ContractElement to a corresponding Solidity ContractBodyElement was supplied.

```
1 convertElementToSolidity (Set StartTime) = setStartTime
```

Listing 4.15: Conversion to Solidity language grammar for StartTime constructor.

The convertToSolidity function will pass each element in the list of ContractElements to the convertElementToSolidity function. Here the specific element will be pattern matched out and converted to the language grammar. Listing 4.15 shows the Set StartTime constructor being pattern matched out and turned into a Solidity language grammar ContractBodyElement with the function shown in 4.16.

```
1 setStartTime :: ContractBodyElement
2 setStartTime = StateVariableElem $ StateVariableDeclaration
3   (ElementaryType uint) [PublicState]
4   (createIdentifier "start") (Nothing)
```

Listing 4.16: Solidity language grammar representation of start time constructor.

After the convertToSolidity function call the contract is now represented in the Solidity language grammar so can be passed to the printSolidity function to be turned into a string representing its valid Solidity syntax and this string is then written to the file path supplied to the outputContract function.

4.5.11 DSL Contract Examples

Bank Example

```
withdrawFromBalance = FunctionElement "withdraw"
  (Financial Withdraw SpecificAmount UserBalance)
depositFromBalance = FunctionElement "deposit"
  (Financial Deposit All UserBalance)
transferFromBalance = FunctionElement "transfer"
  (Financial Transfer SpecificAmount UserBalance)
getBal = FunctionElement "getBalance" (Core GetBalance)

bank = Contract "Bank" [Set Balances, Set Owner,
  withdrawFromBalance, depositFromBalance, transferFromBalance, getBal]
```

Listing 4.17: Bank Contract written in the DSL.

In listing 4.17 we see an implementation of a bank contract in the domain specific language. The bank contract has four functions. The withdraw function specifies that it can be called with a specific amount that the caller would like to withdraw and that they are withdrawing from the balance associated with their address. The deposit function specifies that a user can deposit the entire value of the message sent to their user balance. A transfer function is defined where a user can specify an address where they want to send funds and an amount to send and it will send the entire amount specified from their balance to the receiver's balance. A get balance function is defined using core functionality to allow a user to check their own balance at any time. Finally, an instance of the Contract data type is defined representing the entire bank contract. The four functions defined are passed in along with specifying that the contract needs to set balances for all of its users and set a contract owner. This contract can now be outputted by passing the bank variable representing the contract to the outputContract function and specifying a file path of where the output should go to. When the program is next run, the Solidity file will output to that location. It can be noted that this bank contract was relatively quick and straightforward to write with only a few combinators needed. This is partly because the FinancialFunctionality constructors were built with the common banking functionality needed in mind meaning that this contract in particular is quick to write in the DSL.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.4;
contract Bank {
  constructor () {
    owner = msg.sender;
  }
  mapping (address => uint) balance;
  address payable public owner;
  function withdraw(uint amount) public {
    require(balance[msg.sender] >= amount);
    balance[msg.sender] -= amount;
    payable(msg.sender).transfer(amount);
  }
  function deposit() public payable {
    balance[msg.sender] += msg.value;
  }
  function transfer(address payable _to, uint amount) public payable {
    require(balance[msg.sender] >= amount);
    balance[msg.sender] -= amount;
    balance[_to] += amount;
  }
  function getBalance() public view returns (uint){
    return balance[msg.sender];
  }
}
```

Listing 4.18: Solidity code outputted from bank implementation in DSL.

In the outputted contract shown in listing 4.18 we see that the four functions specified in the DSL correspond to four Solidity functions. The two set variables passed to the contract also correspond with the two contract variables outputted. Also of note is that a constructor has automatically been built initialising the owner of the contract to the contract creator.

Richest Game

A game commonly implemented with smart contracts involves attempting to be the 'richest' participant when the time runs out. Before the time runs out, users can send as much money to the contract as they like. If the amount you send to the contract is the largest single payment sent so far then you become the 'richest' until someone sends a higher payment. If you are the richest when the time runs out, then all of the money that has been sent to the contract becomes yours. Otherwise you win nothing and have lost all of the funds you have sent to the contract. In listing 4.19 we see an

implementation of this game in the domain specific language.

```
richest = Recipient "richest"
highestAmount = UnSignedAmount "highestAmount"

depositToContractBal = Conditioned
  (RequireTime BetweenStartAndEnd)
  (Financial Deposit All ContractBalance)
becomeRichest = Conditioned (RequireVariableRelation
  ValueGreaterThan MessageValue highestAmount) (Join
  (Core (BecomeRecipient (richest)))
  (Core (Update (highestAmount) (MessageValue) [])))
attemptToBecomeRichest = FunctionElement "attemptToBecomeRichest"
  (Join depositToContractBal becomeRichest)
withdrawIfRichest = FunctionElement "withdraw"
  (Conditioned (RequireTime Ended)
  (Conditioned (RequireRecipient richest)
  (Financial Withdraw All ContractBalance)))
checkRichest = FunctionElement "checkRichest"
  (Core (GetVariable (RecipientAddress richest)))

richestGame = Contract "RichestGame" [Set (RecipientAddress (richest)),
  Set ContractBal, Set StartTime, Set EndTime, Set highestAmount,
  attemptToBecomeRichest, withdrawIfRichest, checkRichest]
```

Listing 4.19: Richest Game written in the DSL.

Firstly, a recipient known as the richest is defined. This will be the address of the winner of the game. An unsigned integer called highestAmount is also defined to keep track of the highest single payment sent. A deposit function that allows users to deposit funds to the contract balance while the game is ongoing is defined. A becomeRichest function is defined which checks if the amount of funds in a sent message is greater than the highestAmount sent and if it is, it updates richest to the message sender and updates highestAmount to the new highest sent. The deposit function and becomeRichest function are then combined into one new function called attemptToBecomeRichest using the Join combinator. This function will deposit the funds sent to the contract balance regardless of whether it is the highest amount and will only update highestAmount and richest if the amount sent was indeed the highest amount sent so far. A function to allow the winner to withdraw their earnings after the game has ended is defined along with a function allowing users to check who the current richest is. Finally, an instance of the contract data type is created, passing in the necessary setters and the three functions needed for the contract - attemptToBecomeRichest, withdrawIfRichest and checkRichest. The contract seen in

4.20 is then produced when this contract is sent to the outputContract function.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity ^0.7.4;
3  contract RichestGame {
4    constructor (address payable _richest, uint secondsAfter, uint
       _highestAmount) {
5      richest = _richest;
6      contractBalance = 0;
7      start = block.timestamp;
8      end = start + secondsAfter * 1 seconds;
9      highestAmount = _highestAmount;
10   }
11   address payable public richest;
12   uint public contractBalance;
13   uint public start;
14   uint public end;
15   uint public highestAmount;
16   function attemptToBecomeRichest() payable public {
17     require(block.timestamp >= start && block.timestamp <= end);
18     contractBalance += msg.value;
19     if (msg.value > highestAmount) {
20       richest = msg.sender;
21       highestAmount = msg.value;
22     }
23   }
24   function withdraw() public {
25     require(block.timestamp > end);
26     require(richest == msg.sender);
27     payable(msg.sender).transfer(contractBalance);
28     contractBalance = 0;
29   }
30   function checkRichest() public view returns (address payable){
31     return richest;
32   }
33 }
```

Listing 4.20: Solidity code outputted from Richest Game implementation in DSL.

The three functions defined as withdraw, attemptToBecomeRichest (which is the deposit and becomeRichest functions joined together) and checkRichest have been generated. A start time, end time, contractBalance, highestAmount, and address known as the richest have been defined. A constructor has been generated by the DSL that will assign initial values to the richest, contractBalance, start time, end time and highestAmount when the contract is being deployed.

Auction Example

In listing 4.21 we see the implementation of an auction contract in the DSL. To implement an auction contract, an address variable is needed to keep track of the winner of the auction. An amount representing the highest bid so far and an amount representing a new incoming bid are needed along with a boolean value to keep track of whether or not the winner of the auction has paid the amount promised yet. A bid function is defined which checks that the contest is still ongoing by requiring that the current time is between the start and end point of the auction. It then checks if the newBid amount is greater than the current highest bid. If it is then it updates highestBid and winner to the new current winner and new highest bid placed. In the payIfWinner function, if the auction is over then we allow the winner to pay the full amount they have specified they would pay and when they do, the hasPaid Boolean is set to true to record that the amount has been paid. The checkWinner function is simply a function which allows users to check the address of the winner. The ownerWithdrawAfterAuctionEnded function is allowing the owner of the contract who is the auction holder to withdraw the amount that the winner has deposited to the contract after they have won. Finally, a variable called auction is defined which represents the contract and all functions and contract variables are passed along with specifying that the contract requires the owner address, contract balance, start time and end time to be set also.

```
winner = Recipient "winner"
highestBid = UnSignedAmount "highestBid"
newBid = UnSignedAmount "newBid"
hasPaid = BooleanVariable "hasPaid"

payIfWinner = FunctionElement "payIfWinner"
  (Conditioned (RequireTime Ended) (Conditioned
  (RequireRecipient winner) ((Conditioned
  (RequireVariableRelation ValueEqualTo (highestBid) (MessageValue))
  (Join(Financial Deposit All ContractBalance)
  (Core (Update hasPaid BoolTrue [])))))))
bid = FunctionElement "bid"
  (Conditioned (RequireTime BetweenStartAndEnd)
  (Conditioned (RequireVariableRelation
  ValueGreaterThan newBid highestBid)
  (Join (Core (BecomeRecipient (winner)))
  (Core (Update (highestBid) (newBid) [newBid])))))
checkWinner = FunctionElement "checkWinner"
  (Core (GetVariable (RecipientAddress winner)))
ownerWithdrawAfterAuctionEnded = FunctionElement "withdraw"
  (Conditioned (RequireTime Ended)
  (Conditioned (RequireOwner) (Conditioned (RequireTrue (hasPaid))
  (Financial Withdraw All ContractBalance))))

auction = Contract "Auction" [Set (RecipientAddress (winner)),
  Set Owner, Set ContractBal, Set highestBid, Set hasPaid,
  Set StartTime, Set EndTime, ownerWithdrawAfterAuctionEnded,
  bid, payIfWinner, checkWinner]
```

Listing 4.21: Auction Contract written in the DSL.

When the auction variable is passed to the outputContract function with the output file location specified, the Solidity code shown in listing 4.22 is generated when the program is run. The conversion from DSL sets and function elements to Solidity contract variables and functions is clear.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.4;
contract Auction {
  constructor (address payable _winner, uint _highestBid, bool _hasPaid
      , uint secondsAfter) {
    winner = _winner;
    owner = msg.sender;
    contractBalance = 0;
    highestBid = _highestBid;
```

```
    hasPaid = _hasPaid;
    start = block.timestamp;
    end = start + secondsAfter * 1 seconds;
  }
  address payable public winner;
  address payable public owner;
  uint public contractBalance;
  uint public highestBid;
  bool public hasPaid;
  uint public start;
  uint public end;
  function withdraw() public {
    require(block.timestamp > end);
    require(owner == msg.sender);
    require(hasPaid == true);
    payable(msg.sender).transfer(contractBalance);
    contractBalance = 0;
  }
  function bid(uint newBid) public {
    require(block.timestamp >= start && block.timestamp <= end);
    if (newBid > highestBid) {
      winner = msg.sender;
      highestBid = newBid;
    }
  }
  function payIfWinner() payable public {
    require(block.timestamp > end);
    require(winner == msg.sender);
    if (highestBid == msg.value) {
      contractBalance += msg.value;
      hasPaid = true;
    }
  }
  function checkWinner() public view returns (address payable){
    return winner;
  }
}
```

Listing 4.22: Solidity code outputted from Auction implementation in DSL.

5 Evaluation

5.1 Testing

In order to test the behaviour of the Solidity code outputted by the domain specific language, the Truffle testing framework was used. This allowed tests to be written in JavaScript which could create instances of contracts using user specified constructor values and then call the functions within those contracts and assert that the output achieved correlates to the expected output. Tests were written for all seven of the example contracts implemented in the DSL and bugs discovered with the Solidity code they were compiling to were fixed.

```
kitchind@DESKTOP-SL757SS:/mnt/c/users/kitch$ testrpc
EthereumJS TestRPC v6.0.3 (ganache-core: 2.0.2)

Available Accounts
==================
(0) 0x39a5cadede83c7c7fbe8de03900e1395c8907582
(1) 0x53829c275ce5f337b2ed0e2d30426ab18a33cc47
(2) 0x27a9c5c7a7472b647117322ad3f70f95563dd8ed
(3) 0xeed7f21e574959db65e70055c96401e6bf458e04
(4) 0xc4723016f14d606b2e7e5a7174768d751dc9e9eb
(5) 0x0d3cea654ac055840de0edd0b68ba507f904acea
(6) 0x9db5186266a03c7b5af20209b0612efceb20b936
(7) 0xc8e237ed69adcd61d15269d482786608ccc8de87
(8) 0x8c6465f9c6ffcc4b94623c0d107c11badd521294
(9) 0x69d46b4df8be8dff72a90176e8f262a4ba2dc41d

Private Keys
==================
(0) d8ff61b618f4542580d8a5c9355f4406ef77d29434a86ba701341cdcbc22bfec
(1) 782671fbcc8f10cf43b48db4ee68de8addf0bc8e8bddd5a1cc468798e2dd3ffa
(2) d2e343ef01b9c765aa235eec9bb8338161106c6e5ec961169d1046af8c1d88f5
(3) 5f36714d9f406600b6bc46e7ddc594883ab000775e8a12f862bb28c02464568c
(4) ab062583ca33834eb2831dc6d2f3aafcdbdb76c1ca777d795331b5b24e4ea910
(5) 372f128227fdd7784fb55bd7e6d4e5b7ef5368bc16fc270352b2e1485e14418a
(6) 7ac70aeb7c23e835e691b6ba78e9f07ce77204bdbc2b39f4d30594cb94d6bdf7
(7) 214a425639219fa2181312cf780e106bef9220917025700998350688f49f075f
(8) e434347577f611f7225683e03f5d59388b60120b18f702fa54b5756a2be47ba1
(9) bae21c3861ef66c4f5f62b67e06ef5211dd8869257a12889cd579ee504232be9

HD Wallet
==================
Mnemonic:      music find collect sign off shaft snake fall snack elite common noodle
Base HD Path:  m/44'/60'/0'/0/{account_index}

Listening on localhost:8545
```

Figure 5.1: TestRPC simulating 10 address and 10 private keys.

Ethereum TestRPC was used to simulate an Ethereum blockchain providing unique addresses representing accounts visible to the blockchain. These simulated addresses could then be used when writing tests using the Truffle framework to represent actors in a blockchain.

The tests written for each contract call multiple functions in sequence to ensure that all functions interact with each other as appropriate. For example, when testing the

banking contract, it was important to sequence withdraws, deposits and transfers interlocked.

```
1 it("Joe deposits 10 Wei and then withdraws 7 Wei", function() {
2   return Bank.deployed().then(function(instance) {
3     instance.deposit({from: joe, value: 10});
4     instance.withdraw(7, {from: joe});
5     return instance.getBalance.call({from: joe});
6   }).then(function(balance) {
7     assert.equal(balance.valueOf(), 3, "Joe didn't have a balance of 3"
          );
8   });
9 });
```

Listing 5.1: A unit test for the Bank Contract made with Truffle.

Listing 5.1 shows a unit test simulating an actor called Joe calling the deposit function with 10 Wei, then withdrawing 7 Wei from his account. The test then checks that the balance of Joe's account is equal to 3 and gives an error message if not.

5.2 Analysis of Project Objectives

This project succeeded at designing and implementing an embedded domain specific language in Haskell that could generate secure smart contracts deployable to the EVM. The syntax of the domain specific language is simple to understand and human readable. It accomplished this by taking inspiration from the financial combinators paper. The language is powerful enough to generate many realistic and useful smart contracts, but some functionality is missing that could increase its power. The project has succeeded in demonstrating the usefulness of Haskell for hosting embedded domain specific languages by using Haskell's powerful type system and pattern matching to implement a version of the Solidity Language Grammar in Haskell and also building a domain specific language in Haskell that can translate to the language grammar representation.

5.3 Limitations

5.3.1 High-Level Language Limitations

High-level limitations of the language built include its current lack of support for some composite data types. While the language provides support for lists of addresses, it does not provide support for lists of other types. There is also currently no ability to generate Solidity structs which would allow much more complex contracts to be generated by the language. Also, fallback functions are a core part of Solidity allowing for some action to be performed when a function call fails such as a condition for the call not being reached. Incorporating fallback functionality would have allowed better error handling in the produced contracts. Another high-level improvement that could be made to the DSL would be to further break down some of the data types for a better separated language. For example, the variable data type has 25 different pattern matches but could instead be separated into further data types to avoid this such as separate data types for addresses, amounts, times etc. A disadvantage of the method of implementation used is that the large amount of data types within data types in Haskell means that the language requires a large amount of bracketing to separate out the different arguments of each data constructor and it is easy to make mistakes by misplacing brackets. The Haskell compiler will point out these errors when attempting to run the program and the user then needs to look at exactly how they are placing their brackets to find out what they have done wrong. While the implementation of the entire Solidity language grammar into Haskell makes the code very powerful and reusable, it also means that there is a huge amount of code in the background that needs to be understood at some level for a user to begin expanding the DSL or writing their own DSL using the language grammar. A more simplified version of the language grammar could have made the code base less daunting as many of the language grammar components are not being used by the DSL. A final high-level limitation that I would improve given more time is to have many of the variables being used in functions automatically set without being passed into the contract by the user. For example, if a function is being made proportional to some tax rate using the ProportionalTo constructor then the user should not have to pass in the Set Tax constructors into the contract's list of contract elements as this step could be done automatically during the compile stage when it is seen that TaxRate is being used by some function.

5.3.2 Low-Level Language Limitations

Low-level limitations with the implementation include that the order of elements in the list of contract elements passed to a contract will decide the order of the elements in the outputted contract. This has implications for some scenarios where the user must be aware of how they should pass arguments. For example, if a user is setting both the start and end time variables, they need to always pass the start variable before the end variable because the end variable is assigned based off of the value start has been given. Ideally, there would be an extra check in the language to always put the start assignment in the constructor before the end assignment, regardless of how the list of contract elements is ordered. Another limitation is the way financial functionality is being compiled. Financial functionality is using a lot of predetermined statements that are set in the back end. These provide a good basis for certain functions but allowing the user more customisation over the statements they compile to may have been a better implementation. For example, using the financial functionality Transfer will create a function with parameters such as an address called "_to" representing the address of the person receiving funds. In order to perform some extra action on this "_to" variable that is not part of the predetermined transfer functionality, the user needs to have a good idea of the statements that these financial data types compile to in order to be able to make changes to the variables.

6 Conclusion

6.1 Reflection

I found this project extremely enjoyable to do and feel that I have made a valuable DSL with real world usefulness. From doing this project, I have learned a huge amount about Haskell, domain specific language design and implementation, the Solidity programming language, smart contracts and Ethereum. This project has greatly heightened my interest in blockchain technology and I hope to expand on this work in the future and to work on similar projects utilizing the skills I've learned.

6.2 Future Work

I believe this language could be easily expanded by increasing the number of constructors for each data type. For example, the variables constructor could allow other data types not currently defined such as bytes and arrays. More functionality data constructors could also be added to account for a broader range of scenarios with less emphasis specifically placed on financial functionality. Data constructors allowing users to define structs for example would be very useful as it could allow for complex composite data types keeping track of many variables to be generated such as allowing voting contracts simulating voters casting their ballots to be generated. It would also be interesting to experiment with a more complex top-level language data type. Currently the DSL allows for a single contract at the top-level, however, because Solidity is object oriented it allows for contracts calling other contracts and this would be interesting to experiment with for the DSL. I believe that the methodology used to implement this domain specific language makes a lot of the code reusable for other attempts at creating domain specific languages in Haskell that will compile to Solidity code. The language grammar representation of Solidity along with the printing and abstraction functions makes it incredibly easy to create functions that will take user defined data types and extract information from them to return a corresponding representation in the Solidity language grammar which can then be outputted to a Solidity file. By creating this kind of assembly language to represent Solidity in Haskell I have made this project very reusable to anyone interested in attempting a similar project in the future.

Overall, I am very pleased with the number of features implemented and how many useful contracts it is possible to create using the DSL demonstrated by the sample contracts I have built. While there are some features missing from the language that would be nice to have, the functionality that has been implemented works well, is easy to write and provides a good insight into how Solidity works for a first-time user.

Bibliography

[1] *Functional Programming*, 2021. URL https://wiki.haskell.org/Functional_programming. Accessed: 2020-11-07.

[2] *Embedded domain specific language*, 2021. URL https://wiki.haskell.org/Embedded_domain_specific_language. Accessed: 2020-11-04.

[3] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014. URL https://web.archive.org/web/20180203110042/http://yellowpaper.io/. Accessed: 2020-12-04.

[4] Jake Frankenfield and Erika Rasure. Gas (ethereum). 2014. URL https://www.investopedia.com/terms/g/gas-ethereum.asp#:~:text=What%20Is%20Gas%20(Ethereum)%3F,on%20the%20Ethereum%20blockchain%20platform.

[5] *PROOF-OF-WORK*, 2021. URL https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/#:~:text=Proof%20of%20Work%20(PoW)%20is,difficult%20to%20attack%20or%20overwrite.

[6] *PROOF-OF-STAKE*, 2020. URL https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[7] *Solidity*, 2020. URL https://docs.soliditylang.org/en/v0.8.3/.

[8] S. Sayeed, H. Marco-Gisbert, and T. Caira. Smart contract: Attacks and protections. *IEEE Access*, 8:24416–24427, 2020. doi: 10.1109/ACCESS.2020.2970495.

[9] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: An adventure in financial engineering (functional pearl). *SIGPLAN Not.*, 35(9): 280–292, September 2000. ISSN 0362-1340. doi: 10.1145/357766.351267. URL https://doi.org/10.1145/357766.351267.

[10] *Cardano Documentation*, 2020. URL https://docs.cardano.org/en/latest/.

[11] *Learn about Marlowe*, 2020. URL https://docs.cardano.org/en/latest/marlowe/marlowe-explainer.html.

[12] *Marlowe Tutorial*, 2021. URL https://alpha.marlowe.iohkdev.io/tutorial/index.html#escrow-ex.

[13] M. Wöhrer and U. Zdun. Domain specific language for smart contract development. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2020. doi: 10.1109/ICBC48266.2020.9169399.

[14] Christopher Frantz and Mariusz Nowostawski. From institutions to code: Towards automated generation of smart contracts. pages 210–215, 09 2016. doi: 10.1109/FAS-W.2016.53.

[15] *Language Grammar*, 2021. URL https://docs.soliditylang.org/en/latest/grammar.html.

A1 Appendix

A1.1 Code Location and Run Instructions

The GitHub repository containing all code written for this project and instructions on how to run the code is located at:
`https://github.com/DarrenKitching/Haskell-Smart-Contracts`

A2 Appendix

A2.1 Other Contract Examples

A2.1.1 Shareholders Contract

```
shareHolder = Recipient "shareHolder"
numberOfShares = UnSignedAmount "sharesOwned"

withdrawFromDividends = FunctionElement "withdraw"
  (Financial Withdraw All UserBalance)
depositToContract = FunctionElement "depositToContract"
  (Conditioned (RequireOwner) (Financial Deposit All ContractBalance))
payDividends = FunctionElement "payDividends"
  (Conditioned (RequireOwner) (ProportionalTo (SignedAmount "amount")
  (IndexAddresses (SharesOwned) (Recipient "_to")) (Shares)
  (Financial Transfer SpecificAmount ContractBalance)))
getBalanceAfterPaid = FunctionElement "getBalance" (Core GetBalance)
setSharesOwned = FunctionElement "setShares"
  (Conditioned (RequireOwner)
  (Core (Update (IndexAddresses SharesOwned shareHolder)
  (numberOfShares) [RecipientAddress shareHolder, numberOfShares])))

shareholders = Contract "ShareHolders" [Set ContractBal, Set Owner,
  Set TotalShares, Set SharesOwned, Set Balances,
  withdrawFromDividends, payDividends, depositToContract,
  setSharesOwned, getBalanceAfterPaid]
```

Listing A2.1: Shareholders Contract.

A2.1.2 Taxes Contract

```
withdraw = FunctionElement "customerWithdrawal"
  (Financial Withdraw SpecificAmount UserBalance)
deposit = FunctionElement "customerDeposit"
  (Financial Deposit All UserBalance)
payTaxes = FunctionElement "payTaxes" (ProportionalTo
  (SignedAmount "amount") (IndexAddresses (Balances)
  (Recipient "msg.sender")) TaxRate
  (Financial Transfer SpecificAmount UserBalance))

taxes = Contract "Taxes" [Set Owner, Set ContractBal, Set Balances,
  Set Tax, withdraw, deposit, payTaxes, getBal]
```

Listing A2.2: Taxes Contract.

A2.1.3 Interest Contract

```
customerWithdrawal = FunctionElement "customerWithdrawal"
  (Financial Withdraw SpecificAmount UserBalance)
customerDeposit = FunctionElement "customerDeposit"
  (Financial Deposit All UserBalance)
bankDepositToContract = FunctionElement "depositToContract"
  (Conditioned (RequireOwner) (Financial Deposit All ContractBalance))
payInterest = FunctionElement "payInterest"
  (ProportionalTo (SignedAmount "amount")
  (IndexAddresses (Balances) (Recipient "_to")) InterestRate
  (Financial Transfer SpecificAmount ContractBalance))
payInterestLessDIRT = FunctionElement "payInterestLessDIRT"
  (ProportionalTo (SignedAmount "amount") (IndexAddresses (Balances)
  (Recipient "_to")) InterestRateLessDIRT
  (Financial Transfer SpecificAmount ContractBalance))
transferOwnership = FunctionElement "giveOwnership"
  (Core GiveOwnership)

interest = Contract "Interest" [Set Owner, Set ContractBal,
  Set Balances, Set Interest, Set DIRT, customerWithdrawal,
  customerDeposit, payInterest, payInterestLessDIRT,
  transferOwnership, getBal, bankDepositToContract]
```

Listing A2.3: Interest Contract.

A2.1.4 Lotto Contract

```
lottoWinner = Recipient "winner"
lottoPlayers = AddressList "players"
ticketPrice = UnSignedAmount "ticketPrice"
numberOfPlayers = UnSignedAmount "numberOfPlayers"

buyTicket = FunctionElement "buyTicket"
  (Conditioned (RequireVariableRelation ValueEqualTo
  (MessageValue) (ticketPrice))
  (Join (Financial Deposit SpecificAmount ContractBalance)
  (Join (updatePlayersList) (updateNumberOfPlayers))))
updatePlayersList = (Core (Update (IndexAddressList (lottoPlayers)
  (numberOfPlayers)) (MessageSender) []))
updateNumberOfPlayers = (Core (Update (numberOfPlayers)
  (Increment (numberOfPlayers) (1)) []))
withdrawIfWinner = FunctionElement "withdraw"
  ((Conditioned (RequireRecipient lottoWinner)
  (Financial Withdraw All ContractBalance)))
pickWinner = FunctionElement "pickWinner"
  (Conditioned (RequireNotZero numberOfPlayers)
  ((Conditioned (RequireOwner) (Core (Update
  (RecipientAddress lottoWinner) (IndexAddressList (lottoPlayers)
  (Random numberOfPlayers)) [])))))
checkLottoWinner = FunctionElement "checkWinner"
  (Core (GetVariable (RecipientAddress lottoWinner)))

lotto = Contract "Lotto" [Set (RecipientAddress (lottoWinner)),
  Set numberOfPlayers, Set lottoPlayers, Set ticketPrice,
  Set ContractBal, Set Owner, buyTicket, pickWinner,
  withdrawIfWinner, checkLottoWinner]
```

Listing A2.4: Lotto Contract.

A3 Appendix

A3.1 Tests

A3.1.1 Auction Contract Tests

```
const Auction = artifacts.require("Auction");

contract("Auction", function (accounts) {
    const mary = accounts[0];
    const joe = accounts[1];
    const bill = accounts[2];
    const julie = accounts[3];
    it("Auction test", function() {
      return Auction.deployed().then(function(instance) {
        instance.bid(1000, {from: mary});
        instance.bid(1100, {from: joe});
        instance.bid(500, {from: mary});
        instance.bid(1150, {from: bill});
        return instance.checkWinner.call();
      }).then(function(winner) {
        assert.equal(winner, bill, "Bill was not the winner.");
      });
    });
    return assert.isTrue(true);
});
```

Listing A3.1: Tests written for Auction Contract.

A3.1.2 Bank Contract Tests

```
const Bank = artifacts.require("Bank");

contract("Bank", function (accounts) {
    console.log("Accounts visible: ", accounts);
    const mary = accounts[0];
    const joe = accounts[1];
    const bill = accounts[2];
    const julie = accounts[3];
    it("Put 5 Wei into Mary's account", function() {
      return Bank.deployed().then(function(instance) {
        instance.deposit({from: mary, value: 5});
        return instance.getBalance.call({from: mary});
      }).then(function(balance) {
        assert.equal(balance.valueOf(), 5, "Mary didn't have a balance
           of 5");
      });
    });
    it("Joe deposits 10 Wei and then withdraws 7 Wei", function() {
      return Bank.deployed().then(function(instance) {
        instance.deposit({from: joe, value: 10});
        instance.withdraw(7, {from: joe});
        return instance.getBalance.call({from: joe});
      }).then(function(balance) {
        assert.equal(balance.valueOf(), 3, "Joe didn't have a balance
           of 3");
      });
    });
    it(`Bill deposits 15 Wei. He transfer 5 Wei to Julie.
        Bill's balance should now be 10 and Julie's balance should be
           5.`, function() {
      return Bank.deployed().then(function(instance) {
        instance.deposit({from: bill, value: 15});
        instance.transfer(julie, 5, {from: bill});
        return instance.getBalance.call({from: bill});
      }).then(function(balance) {
        assert.equal(balance.valueOf(), 10, "Bill didn't have a balance
            of 10");
      });
    });
    return assert.isTrue(true);
});
```

Listing A3.2: Tests written for Bank Contract.

A3.1.3 Shareholders Contract Tests

```
const ShareHolders = artifacts.require("ShareHolders");

contract("ShareHolders", function (accounts) {
    const mary = accounts[0];
    const joe = accounts[1];
    const bill = accounts[2];
    const julie = accounts[3];
    it("Pay out Dividends to Mary", function() {
      return ShareHolders.deployed().then(function(instance) {
        instance.depositToContract({value: 1000});
        instance.setShares(mary, 10);
        instance.payDividends(mary, 200, {value: 200})
        return instance.getBalance.call({from: mary});
      }).then(function(balance) {
        assert.equal(balance.valueOf().toNumber(), 10,
          "Mary didn't have a balance of 5");
      });
    });
    return assert.isTrue(true);
});
```

Listing A3.3: Tests written for Shareholders Contract.

A3.1.4 Taxes Contract Tests

```
const Taxes = artifacts.require("Taxes");

contract("Taxes", function (accounts) {
  const mary = accounts[0];
  const joe = accounts[1];
  const bill = accounts[2];
  const julie = accounts[3];
  const taxCollector = accounts[4];
  it("Collect taxes", function() {
    return Taxes.deployed().then(function(instance) {
      instance.customerDeposit({from: mary, value: 1000});
      instance.payTaxes(taxCollector, 0, {from: mary});
      return instance.getBalance.call({from: mary});
    }).then(function(balance) {
      assert.equal(balance.valueOf().toNumber(), 750,
        "Mary didn't have a balance of 750");
    });
  });
});
```

Listing A3.4: Tests written for Taxes Contract.

A3.1.5 Interest Contract Tests

```
const Interest = artifacts.require("Interest");

contract("Interest", function (accounts) {
  const mary = accounts[0];
  const joe = accounts[1];
  const bill = accounts[2];
  const julie = accounts[3];
  it("Pay interest to Mary", function() {
    return Interest.deployed().then(function(instance) {
      instance.customerDeposit({from: mary, value: 1000});
      instance.depositToContract({value: 100});
      instance.payInterest(mary, 100);
      return instance.getBalance.call({from: mary});
    }).then(function(balance) {
      assert.equal(balance.valueOf().toNumber(), 1020,
        "Mary didn't have a balance of 1020");
    });
  });
  it("Pay interest less DIRT to Joe", function() {
    return Interest.deployed().then(function(instance) {
      instance.customerDeposit({from: joe, value: 1000});
      instance.depositToContract({value: 100});
      instance.payInterestLessDIRT(joe, 100);
      return instance.getBalance.call({from: joe});
    }).then(function(balance) {
      assert.equal(balance.valueOf().toNumber(), 1017,
        "Joe didn't have a balance of 1017");
    });
  });
  return assert.isTrue(true);
});
```

Listing A3.5: Tests written for Interest Contract.

A3.1.6 Richest Game Contract Tests

```
const RichestGame = artifacts.require("RichestGame");

contract("RichestGame", function (accounts) {
  const mary = accounts[0];
  const joe = accounts[1];
  const bill = accounts[2];
  const julie = accounts[3];
  it("Play the Richest Game", function() {
    return RichestGame.deployed().then(function(instance) {
      instance.attemptToBecomeRichest({from: mary, value: 1000});
      instance.attemptToBecomeRichest({from: joe, value: 600});
      return instance.checkRichest.call({from: mary});
    }).then(function(richestPerson) {
      assert.equal(richestPerson, mary, "Mary was not the richest
          person.");
    });
  });
  it("Play the Richest Game", function() {
    return RichestGame.deployed().then(function(instance) {
      instance.attemptToBecomeRichest({from: mary, value: 1000});
      instance.attemptToBecomeRichest({from: joe, value: 600});
      instance.attemptToBecomeRichest({from: bill, value: 3000});
      instance.attemptToBecomeRichest({from: julie, value: 2150});
      return instance.checkRichest.call();
    }).then(function(richestPerson) {
      assert.equal(richestPerson, bill, "Bill was not the richest
          person.");
    });
  });
  return assert.isTrue(true);
});
```

Listing A3.6: Tests written for Richest Game Contract.

A3.1.7 Lotto Contract Tests

```
const Lotto = artifacts.require("Lotto");

contract("Lotto", function (accounts) {
  console.log("Accounts visible: ", accounts);
  const mary = accounts[0];
  const joe = accounts[1];
  const bill = accounts[2];
  const julie = accounts[3];
  it("Lotto run with four participants", function() {
    return Lotto.deployed().then(function(instance) {
      instance.buyTicket(5, {from: mary, value: 5});
      instance.buyTicket(5, {from: joe, value: 5});
      instance.buyTicket(5, {from: bill, value: 5});
      instance.buyTicket(5, {from: julie, value: 5});
      instance.pickWinner();
      return instance.checkWinner.call();
    }).then(function(winner) {
      if(winner == mary) {
        assert.equal(winner, mary, "Mary was not the winner.");
      }
      else if (winner == joe) {
        assert.equal(winner, joe, "Joe was not the winner.");
      }
      else if (winner == bill) {
        assert.equal(winner, bill, "Bill was not the winner.");
      }
      else if (winner == julie) {
        assert.equal(winner, julie, "Julie was not the winner.");
      }
      else {
        assert.equal(winner, mary,
          "Neither Mary, Joe, Bill nor Julie was the winner. Something
             went wrong!")
      }
    });
  });
  return assert.isTrue(true);
});
```

Listing A3.7: Tests written for Lotto Contract.