

MENG INDIVIDUAL PROJECT - PROJECT REPORT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

SmartFin - Implementing a Financial Domain-Specific Language for Smart Contracts

Author:
Daniel Dean

Supervisor:
Prof. Susan Eisenbach

June 28, 2019

Abstract

In the world of financial engineering, the use of smart contracts - programs which run on blockchain platforms and deal with cryptocurrencies - is becoming widespread. The implementation of financial contracts in the form of smart contracts is useful for automating the payments they obligate, and for keeping track of financial dealings for auditing purposes. Unfortunately, due to the high complexity of existing smart contract languages, these smart contracts are difficult to implement and evaluate, and can often contain vulnerabilities.

Long before blockchains became commonplace, a functional domain-specific language for the declaration of financial contracts was created by Simon Peyton Jones, Jean-Marc Eber, and Julian Seward[27]. This DSL was devised to make financial contracts less verbose and complex, easier to evaluate mathematically, and less error-prone.

We present a smart contract implementation of a slightly modified version of the aforementioned DSL, called *SmartFin*. A single smart contract is implemented that can represent any SmartFin financial contract passed into its constructor. We also present a web client for composing and evaluating SmartFin financial contracts, and deploying and interacting with their corresponding smart contract instances. As such, a user need only compose a SmartFin financial contract, and is not required to program a smart contract. This allows smart contract representations of financial contracts to be created and evaluated with ease, and the simple syntax and semantics of SmartFin make implementation errors significantly less common in comparison to writing a bespoke smart contract.

Acknowledgements

During this huge undertaking, I have been guided and supported along the way by a great many people; for providing their support, I'd like to express my deepest thanks to the following:

- To Professor Susan Eisenbach and Professor Sophia Drossopoulou, for the great deal of feedback, encouragement, and advice that they have imparted to me throughout this project.
- To Dr. Tony Field, for helping me start off on the right foot with some practical advice at the project's beginnings.
- To Dr. Panos Parpas, for lending me some of his invaluable expertise in the field of financial engineering.
- To Ioannis and Zubair, for weathering the trenches with me for this journey.
- And lastly, to my family, for making me the person I am today.

Contents

1	Introduction	7
1.1	Financial Contracts	7
1.2	A Combinator Domain-Specific Language to Represent Contracts	7
1.3	Smart Contracts	8
1.4	Our Contributions	8
1.5	Challenges	10
2	Background	11
2.1	Overview	11
2.2	Ethereum, and Smart Contracts	11
2.2.1	Ethereum	11
2.2.2	Smart Contracts	12
2.2.3	Why Ethereum?	12
2.3	The Design of the Original Combinator DSL for Financial Contracts	13
2.3.1	Overview	13
2.3.2	Initial Combinators	13
2.3.3	Example Financial Contracts in the Original DSL	15
2.3.4	Representing Financial Contracts	16
2.3.5	Difficult Combinators	16
2.3.6	Payment	17
2.3.7	Handling Observables	17
2.4	Development Tools	18
2.5	SmartFin Smart Contract Implementation	19
2.5.1	Creating a Library for an Existing Smart Contract Language	20
2.5.2	Passing a SmartFin Contract to a Smart Contract	21
2.5.3	Creating a Bespoke Language and Compiling to a Low-Level Language	22
2.5.4	SmartFin Implementation Summary	22

2.6	External Tool for Financial Smart Contract Interaction	23
2.7	SmartFin Contract Cost Analysis	23
2.8	Remarks	24
3	The Design of SmartFin	25
3.1	Overview of SmartFin Financial Contracts	25
3.2	Syntax	25
3.3	Semantics	26
3.4	Remarks	27
4	Smart Contract Design and Implementation	28
4.1	Requirements	29
4.2	Design	29
4.2.1	High Level Design Challenges	29
4.2.2	Financial Smart Contract ABI	32
4.3	Implementation	34
4.3.1	Technologies	34
4.3.2	Smart Contracts in Rust	35
4.3.3	ABI Methods	35
4.3.4	Storage	37
4.4	Evaluation	40
4.4.1	Design of the Smart Contract's ABI	40
4.4.2	The Storage Module	42
4.4.3	Automated Testing	42
4.4.4	Conclusion	44
4.5	Remarks	44
5	Designing and Implementing SmartFin Combinators	45
5.1	Designing a Programmatic Representation of SmartFin Combinators	46
5.1.1	General Representation of Combinators	46
5.1.2	ContractCombinator Methods	46
5.2	Implementation	48
5.2.1	The ContractCombinator Module	48
5.2.2	Simple ContractCombinator Implementations	49
5.2.3	The ZeroCombinator and OneCombinator Modules	49
5.2.4	The AndCombinator Module	49
5.2.5	The OrCombinator Module	50

5.2.6	The GiveCombinator Module	50
5.2.7	The ScaleCombinator Module	50
5.2.8	The TruncateCombinator Module	51
5.2.9	The ThenCombinator Module	51
5.2.10	The GetCombinator Module	51
5.2.11	The AnytimeCombinator Module	51
5.3	Evaluation	52
5.3.1	Combinator Representation	52
5.3.2	Design of the ContractCombinator Trait	52
5.3.3	Design of the Combinators' Implementations	53
5.3.4	Testing	54
5.3.5	Conclusion	55
5.4	Remarks	55
6	The Financial Smart Contract Web Client	56
6.1	The Design of the Web Client	57
6.1.1	Architecture	57
6.1.2	Web Client Components	57
6.1.3	Other Functionality	58
6.2	Implementation	58
6.2.1	Technologies	58
6.2.2	Contract Utility Functions	58
6.2.3	Main View	61
6.2.4	Blockchain Connection View	62
6.2.5	Main-Menu View	63
6.2.6	Contract Composition View	63
6.2.7	Contract Monitoring View	64
6.2.8	Contract Evaluation	65
6.3	Evaluation	69
6.3.1	Web Client Design	69
6.3.2	Web Client Functionality	70
6.3.3	Evaluator Design	71
6.3.4	Testing	71
6.3.5	Conclusion	73
6.4	Remarks	73

7	Evaluation	74
7.1	Individual Chapter Evaluations	74
7.2	User Feedback	75
7.3	Solidity Case Studies	76
7.3.1	Simple Contract	76
7.3.2	European Option	78
7.3.3	Loan with Variable Repayment	79
7.3.4	Case Study Conclusions	81
7.3.5	Case Study Remarks	81
7.4	Remarks	82
8	Conclusions	83
8.1	Reflections	84
8.2	Areas for Future Work	84
8.2.1	Maximal SmartFin Contract Evaluation and Numerical Modelling	84
8.2.2	Expansion of the SmartFin DSL	85
A	SmartFin Smart Contract ABI	86
B	Solidity Implementation of the <i>One</i> Case Study	88
C	Solidity Implementation of the <i>European Option</i> Case Study	91
D	Solidity Implementation of the <i>Loan with Variable Repayment</i> Case Study	95
E	User Manual	99
E.1	Getting Started	99
E.1.1	Overview	99
E.1.2	Installation	99
E.2	SmartFin - The Financial Contract DSL	100
E.2.1	Overview	100
E.2.2	Combinators	100
E.2.3	Examples	102
E.3	Connecting to a Blockchain	103
E.3.1	Overview	103
E.3.2	Manual Connection	103
E.4	Main Menu	104
E.5	Composing a SmartFin Contract	105

E.5.1	Composing a SmartFin Financial Contract	105
E.5.2	Deploying a Financial Smart Contract	106
E.6	Monitoring a Financial Smart Contract	107
E.6.1	Financial Smart Contract Details	107
E.6.2	Interacting with a Financial Smart Contract	108
E.7	Evaluating a SmartFin Financial Contract	111
E.7.1	Top-level Acquisition Time	111
E.7.2	Anytime Acquisition Time	111
E.7.3	Or Choices	112
E.7.4	Value	113

Chapter 1

Introduction

1.1 Financial Contracts

In the real world, two parties may create a *financial contract* to describe a set of payments to be made between them under certain conditions. These contracts can suffer from numerous issues in real world usage; for one thing, the financial world is plagued with jargon, which can make contracts difficult to interpret, and complicated to compose. This can lead to unneeded verbosity, where certain terms may be repeated multiple times in a single contract. Traditional financial contracts can also be difficult to analyse mathematically in terms of potential cost or value, due to their complexity and potential ambiguity - resulting in the appearance of unnoticed errors in contract definitions.

1.2 A Combinator Domain-Specific Language to Represent Contracts

A hypothetical solution to alleviate some of these issues has been proposed by Simon Peyton Jones, Jean-Marc Eber, and Julian Seward in the paper *Composing Contracts: An Adventure in Financial Engineering*[27]. Their proposed solution employs a combinator domain-specific language, which is a type of functional programming language where specific terms, i.e. *combinators*, can be composed to produce a program. This solution involves using combinators to represent financial contracts. This is possible as these financial contracts can typically be described such that they are composed of smaller contracts - i.e. *sub-contracts*. These combinators can be as simple as `one`, which requires the counter-party to pay a single unit of a given currency to the owner. They can also describe transformations on inner combinators, such as `scale`, which multiplies any monetary values in inner combinators by a given value. The contract `scale(5, one(GBP))` will therefore require the counter-party to pay the owner £5.

The definition of financial contracts using a combinator DSL has numerous upsides. For one thing, even with few combinators a contract writer can define a huge variety of financial contracts, thus reducing the amount of jargon required. If any contracts which cannot be represented by the combinator DSL are found, new combinators can easily be added due to their modular nature. Programmatic definition of financial contracts can also cut down on needless repetition, and there is no room for interpretation. Additionally, the use of combinators facilitates mathematical analysis of financial contracts' values, by the composable nature of these values - where each combinator's value is a function of their inner combinators' values. While there is an implementation of an evaluation process for financial contracts written in the DSL, there is no programmatic implementation of these contracts.

1.3 Smart Contracts

Since the original DSL was first described, cryptocurrencies have proliferated; because of this, it has become popular to represent financial contracts using *smart contracts*. Smart contracts are programs which can be deployed to a blockchain, and then called at any time to execute some specified code. One specific functionality they provide is the ability to obtain and transfer cryptocurrencies, thus facilitating payments between multiple parties under specified conditions in an automated manner[18]. Writing smart contract representations of financial contracts allows financial institutions to use blockchains between institutions for payment of funds according to these financial contracts, or simply to keep track of existing financial contracts for auditing purposes thanks to the immutability of blockchains.

While existing smart contract languages can provide a strict and unambiguous smart contract representation of a financial contract, i.e. a *financial smart contract*, there are issues with this manner of implementation. One such issue is that many smart contract languages are exceedingly error-prone, meaning that it is very possible to write a financial smart contract with unintended consequences - in fact, it has been estimated that 45% of smart contracts on the Ethereum blockchain (a platform for hosting smart contracts) contain vulnerabilities[30].

Take the smart contract code in listing 1.1, written in Solidity (a programming language for smart contracts); this code simply transfers funds from the smart contract to the caller, and then decrements a *balance* representing the funds the caller can withdraw. To the untrained eye (and the Solidity compiler), this code snippet contains no errors; in actuality, it contains one of the most severe kinds of vulnerabilities, allowing a reentrancy attack from a malicious user. A reentrancy attack is where funds are transferred before a function’s invocation is finished, allowing the function to be called again before the transfer is complete and the balance is decremented (as transfers can trigger function calls)[10]. This can result in the smart contract being drained of all of its funds. This is an example of a severe vulnerability that can be easy to miss when implementing smart contracts in a smart contract language.

```
1 function withdrawOneWei() public {  
2     msg.sender.call.value(1);  
3     balances[msg.sender] = balances[msg.sender] - 1;  
4 }
```

Listing 1.1: A Solidity function which is vulnerable to a reentrancy attack¹.

Smart contracts are also difficult to analyse mathematically in terms of definitive costs, as most smart contract languages are relatively complex (in comparison to the combinator DSL mentioned earlier). Complex functionality like iteration and recursion can make the outcome of program execution difficult to evaluate, and also makes errors relatively easy to introduce and difficult to discover. These features are not required for the representation of financial contracts, as demonstrated by their absence from the aforementioned DSL. Overall, the issues mentioned here make smart contract languages a risky choice when creating financial smart contracts, as it can be easy to allow erroneous behaviour to occur, and difficult to find such errors by analysis.

1.4 Our Contributions

In this work, we present several contributions with the aim of improving the ease of implementation and reducing the risk of erroneous behaviour for financial smart contracts:

¹Solidity syntax highlighting obtained from the `solidity-latex-highlighting` package written by Sergei Tikhomirov, used under the MIT license, available at <https://github.com/s-tikhomirov/solidity-latex-highlighting>.

1. **SmartFin:** A combinator DSL for representing financial contracts, derived from the *original DSL* created by Peyton Jones et al. [27], with slight modifications to enable a smart contract implementation. The design of SmartFin is detailed in chapter 3.
2. **SmartFin Smart Contract Implementation:** An Ethereum-compatible smart contract that can represent any given SmartFin financial contract as a financial smart contract. The implementation of this smart contract is detailed in chapters 4 and 5.
3. **Web Client:** A web client for managing SmartFin smart contracts. The implementation of the web client is detailed in chapter 6. The web client implements the following functionality:
 - Composition of SmartFin financial contracts, with syntax verification and detailed error reporting with stack traces.
 - Evaluation of SmartFin financial contracts in a step-by-step manner, to calculate the value that a contract is worth given all required external input (provided by the user) at each required step. The times that all evaluated payments would occur can also be displayed.
 - Deployment of SmartFin financial smart contracts to any compatible blockchain.
 - Monitoring of any deployed SmartFin financial smart contracts state, and displaying this state to a user.
 - Interaction with deployed SmartFin financial smart contracts, allowing users to provide any input that the financial contract requires.

With all of these tools together, a user can define a financial contract in the SmartFin DSL using the web client; this SmartFin financial contract can be evaluated in the web client in a step-by-step manner, or can be passed to the implemented smart contract and deployed to a connected compatible blockchain. The deployed financial smart contract can be monitored and interacted with through the web client, and as such all of the required functionality to use SmartFin financial smart contracts is available in the web client. The implemented smart contract can take any given SmartFin contract definition in its constructor and modify its state so that its behaviour matches the given financial contract's behaviour, by implementing logic for all combinators' semantics. A dependency graph of the contributions implemented is depicted in figure 1.1.

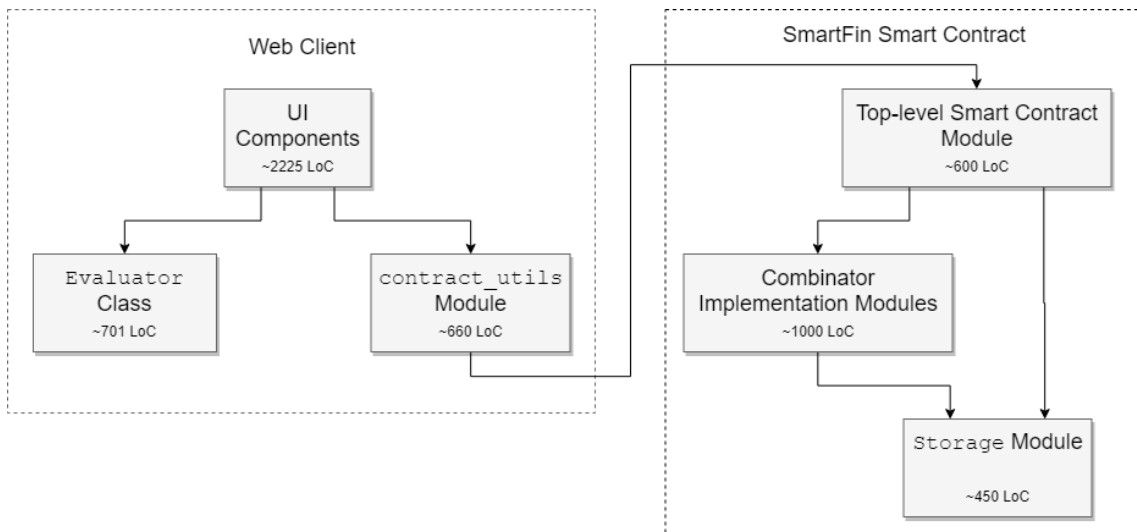


Figure 1.1: A dependency graph of the modules implemented for this project, with their approximate *Lines of Code* (not including tests).

1.5 Challenges

SmartFin Smart Contract Implementation

Implementing the SmartFin combinator DSL in a smart contract is no small feat; while the number of combinators is relatively limited, representing *any* set of combinators requires a robust generic design when it comes to the implementation of the combinators' semantics and state. Furthermore, this design needed be applicable to *all* combinators, and some are quite unintuitive to represent programmatically. The representation of SmartFin combinators is described in [chapter 5](#).

Ethereum Limitations

Due to limitations of the Ethereum platform, it is not possible to create a one-to-one representation of a SmartFin contract in smart contract form. Designing solutions to the issues caused by the nature of the Ethereum platform required compromises to be made, and minimising the resulting compromises required designing multiple alternative solutions and evaluating them objectively. Solutions to issues stemming from the Ethereum platform are discussed in [chapter 4](#).

SmartFin Contract Evaluation

Evaluating SmartFin financial contracts is not a simple problem, even when approaching it in a step-by-step manner, for a couple of reasons. One issue is that SmartFin contracts deal heavily with time, making time an important factor in the evaluation of SmartFin contracts. Keeping track of distinct periods of time based on the SmartFin contract and user interaction was required for step-by-step analysis, requiring a system of describing these time periods to be designed and implemented. Furthermore, keeping track of the current state while requiring the user to enter data can be quite complex, and the ability for the user to revert to any earlier state of step-by-step evaluation can make this even worse. Additionally, the evaluation process requires backtracking for certain combinators, thus resulting in even more complicated behaviour. The implementation of step-by-step evaluation is described in [section 6.2.8](#).

Chapter 2

Background

2.1 Overview

This chapter describes the background research which has been carried out, with regards to Ethereum and Ethereum smart contracts, the original financial contract DSL, the different ways that SmartFin can be implemented, how observable data can be handled, how the cost of financial contracts written in SmartFin can be evaluated, and how tools for interacting with financial smart contracts can be implemented.

2.2 Ethereum, and Smart Contracts

In order to write a smart contract implementation of SmartFin, it is necessary to understand the inner workings of smart contracts and the platforms which host them.

2.2.1 Ethereum

Ethereum is a platform which hosts a cryptocurrency called *Ether* - this platform uses a custom blockchain (a data structure designed to model an immutable append-only ledger) which is hosted on a distributed set of nodes. Users of the platform have a certain amount of Ether, which acts currency. Ether can be exchanged between users through transactions, which are stored on the blockchain. Ethereum operates similarly to other typical cryptocurrency platforms, like Bitcoin, in this sense[33].

A blockchain consists of several blocks. These blocks contain all of the data in the platform, including any transactions which have taken place. Blocks can only be appended to a blockchain, and existing blocks cannot be modified. In order to add new blocks to the blockchain, a number which fulfils specified conditions (a *cryptographic nonce*) must first be found by a node in the system - nodes who search for a nonce are called *miners*. The important detail is that finding a nonce requires a certain amount of work, which is dependent on a *difficulty*. This difficulty can dynamically change to meet the needs of the platform - for example, the difficulty of mining Bitcoin blocks is scaled to modulate the average number of blocks mined in an hour[33]. When a miner discovers a valid nonce, they can create a new block containing any pending transactions and append it to the blockchain[18]. In an Ethereum blockchain, a successful miner is rewarded with some Ether - this is one method of introducing new currency into the platform, and encourages mining to occur. If a miner discovers a valid nonce and thus attempts to append a block to the blockchain, all other nodes will validate the block before accepting it as legitimate, thus ensuring that a consensus is reached on the validity of new blocks[33].

2.2.2 Smart Contracts

Ethereum differs from many other blockchain platforms by also providing a platform upon which special programs (i.e. *smart contracts*) can be run. These programs are also stored on the blockchain, and have functions which can be called by any user of the platform, or by other smart contracts. Smart contracts can also transfer Ether through transactions.

Smart contracts typically run on the *Ethereum Virtual Machine*, or *EVM* for short, which runs on many nodes across the network. When a smart contract is run on a node which also creates the next block, the node is reimbursed for the computational resources in *gas* by the account that called the executed function. The gas fee for a contract is an amount of Ether that is proportional with the computational resources used to run the contract; more computationally intense contracts will have higher gas fees.

The execution of a smart contract must be deterministic, so that all nodes in the network agree on the outcome of the program's execution - and thus agree on whether any given block is valid. This is not any more efficient than traditional execution, and actually uses *more* resources due to repeated execution of the same program; instead, the benefits of smart contracts include extreme fault-resistance (as only one live node is required to run the program), immutability of any results stored on the blockchain, and a lack of any trusted third-parties[18].

The trustless, fault-resistant, immutable nature of smart contracts makes them a great candidate for hosting financial contracts. If the contract required a trusted third-party, then the outcome could be fraudulent if the third-party were biased. If smart contract execution weren't fault-resistant, then there would be no guarantee that any required payments which are possible to fulfil ever actually occur. If the execution were reversible or mutable, then the contracts would become pointless as either party could modify them as desired.

Unfortunately, smart contracts are not perfect. A major issue with these programs is that they can often contain hard-to-spot vulnerabilities which can result in smart contracts making unintended transactions, and thus potentially costing the owner large sums of money. These issues are not uncommon - in fact, it was estimated in 2016 that around 45% of smart contracts contain vulnerabilities[30].

One cause of this issue is that high-level smart contract languages typically provide a complex set of features, including difficult-to-use functionality like iteration and recursion. By giving the developer more freedom to implement complicated smart contract behaviour instead of restricting the use of error-prone features, the risk of introducing errors - like the reentrancy vulnerability demonstrated earlier in listing 1.1 - is increased. Even cutting-edge smart contract languages designed with the prevention of common vulnerabilities in mind, like Flint[35], still provide a complex feature set and thus can't prevent the introduction of vulnerabilities by human error.

This is the main motivator behind the project; allowing financial contract authors to use a restricted yet versatile DSL to write financial contracts and providing a generic smart contract implementation enables contract costs to be evaluated mathematically, and reduces the risk of the financial contract author introducing vulnerabilities or erroneous behaviour.

2.2.3 Why Ethereum?

While Ethereum *does* implement the smart contract functionality required for implementing Smart-Fin's behaviour, it is not the only platform to do so. Many other platforms also offer some level of smart contract functionality, including Bitcoin[29]. One reason for choosing Ethereum is that there exists a large set of development tools available for supporting Ethereum projects - some of which are described in section 2.4. Ethereum also has a large community of users, resulting in a greater

volume of documentation compared to other smart contract platforms. The sizeable user-base also means that SmartFin will be useful to more people than if it were implemented on a less popular platform. Besides these factors, Ethereum smart contracts are also more user-friendly than many other platforms (like Bitcoin Script), making the task of implementation less difficult. For all of these reasons, Ethereum makes a good platform for the implementation of SmartFin.

2.3 The Design of the Original Combinator DSL for Financial Contracts

SmartFin, the proposed DSL for financial contracts, is based on a DSL described by Peyton Jones et al.[27]. The design of the original DSL will not be modified significantly, and any modifications are noted clearly in section 3.3.

2.3.1 Overview

The original financial contract DSL is a combinator language used to describe financial contracts. A contract can be represented by a single combinator, or by some composition of combinators. Each financial contract has a *holder*, and a *counter-party*. Typically, the counter-party will be the party making payments, and the holder will be the party receiving payments.

A financial contract written in the original DSL can be *acquired* by the holder at any point in time, but the responsibilities of each party may differ depending on when the contract is acquired. For example, consider a contract C_1 which requires the counter-party to pay the holder £100 on noon of January 1st 2019 and again on noon of January 1st 2020. C_1 requires 2 payments to occur if acquired before 12:00 on 01/01/19, 1 payment to occur if acquired by 12:00 01/01/20, or no payments to occur if acquired after this point. The acquisition date of a contract will therefore affect the responsibilities and thus the value of the contract for each party.

A financial contract written in the original DSL may *expire* if no responsibilities outlined in the contract take effect when the contract is acquired after a certain time. For example, the contract C_1 has no effect if acquired after 12:00 on 01/01/20. This date is called the *horizon* of the contract. An important thing to note is that a contract's responsibilities could potentially extend past the contract's horizon, but a contract acquired after its horizon will have no effect.

Some financial contracts written in the DSL may be dependent on certain parameters. The contract C_1 , for instance, defines payments of a specific amount on two specific dates. This contract would need to be defined with a constant representing £100, and two date/times. A contract could also be dependent on an variable value, such as the average temperature in London in Celsius, or the distance between two people in metres. Such an objective numeric value is called an *observable*, and can be constant or varying over time. In the original DSL, it is possible to perform arithmetic and functions on observables, values, and dates[27].

2.3.2 Initial Combinators

The set of combinators defined in the original DSL is described below, along with each combinator's type signature (described using the notation of Haskell). These combinator definitions originate from Peyton Jones et al.'s paper on the original DSL[27]. The notation used for describing the original DSL is defined in table 2.1.

c, d	<i>Contract</i>
o	<i>Observable</i>
t	<i>Date/Time</i>
k	<i>Currency</i>

Table 2.1: Conventions for the Original DSL’s Description

`zero :: Contract`

This combinator represents a contract with no terms. It can be acquired at any time, and thus has no horizon.

`one :: Currency -> Contract`

This combinator represents a contract which requires the counter-party to immediately pay the holder one unit of the given currency. This contract can be acquired at any time, and thus has no horizon.

`give :: Contract -> Contract`

`give(c)` represents `c` with all responsibilities reversed (e.g. if the holder acquires `give(one(k))`, they must pay the counter-party 1 unit of currency `k` immediately). The horizon of `give(c)` is the same as the horizon of `c`.

`and :: Contract -> Contract -> Contract`

When `and(c, d)` is acquired, both `c` and `d` are acquired immediately. Expired sub-contracts are not acquired. `and(c, d)`’s horizon is the latest of `c` and `d`’s horizons.

`or :: Contract -> Contract -> Contract`

When `or(c, d)` is acquired, the holder immediately acquires either `c` or `d`. If one has expired, the holder cannot acquire it (and must acquire the other if possible). The horizon of `or(c, d)` is the latest of `c` and `d`’s horizons.

`truncate :: Date -> Contract -> Contract`

When `truncate(t, c)` is acquired, the holder acquires `c`. The horizon of `truncate(t, c)` is the earliest of `t` and the horizon of `c` (thus `truncate(t, c)` cannot be acquired after either horizon has passed).

`then :: Contract -> Contract -> Contract`

When acquiring `then(c, d)`, the holder acquires `c` if `c` has not expired, or `d` if `c` has expired and `d` has not. `then(c, d)`’s horizon is the latest of `c` and `d`’s horizons.

```
scale :: Observable -> Contract -> Contract
```

`scale(o, c)` represents `c` with all payments multiplied by the value of the observable `o` at the time of acquisition. `scale(o, c)` has the same horizon as `c`.

```
get :: Contract -> Contract
```

After the holder acquires `get(c)`, `c` is acquired at the moment in time when the horizon of `c` is reached. `get(c)` has the same horizon as `c`.

```
anytime :: Contract -> Contract
```

After `anytime(c)` is acquired, `c` must be acquired by the holder at any point before it expires. The holder can decide when to acquire `c`. `anytime(c)` has the same horizon as `c`.

2.3.3 Example Financial Contracts in the Original DSL

European Option

A European option allows the holder to choose, on a particular date, between acquiring an underlying contract or acquiring nothing. A European option allowing the holder to choose between receiving £500 GBP from the counter-party, and receiving/paying nothing, at midnight on the 1st of January 2020 is defined as follows:

```
get(truncate(<01/01/2020, 00:00:00>, or(  
    scale(500, one(GBP)),  
    zero  
)))
```

The `get` combinator, if acquired, causes the sub-contract to be acquired at its horizon. This means that `truncate(<01/01/2020 00:00:00>, or(scale(500, one(GBP)), zero))` is acquired at 00:00:00 on 01/01/2020, as the `truncate` combinator sets the sub-contract's horizon to this time.

The `or` combinator is thus acquired at midnight on the 1st of January 2020, as long as the contract is acquired before this. This enables the holder to choose to acquire one of the two sub-contracts at this time. The first sub-contract is `scale(500, one(GBP))`. The `one(GBP)` combinator requires the counter-party to pay the holder £1 upon acquisition, and the `scale(500...` combinator multiplies this by £500, thus this sub-contract requires the counter-party to pay the holder £500 in total upon acquisition. The second sub-contract, `zero`, requires neither the holder nor the counter-party to pay anything. As such, the holder can choose between being paid £500 or £0 at this time.

Simple Loan with Variable Repayment

Take a contract which can only be acquired before 00:00:00 on 01/01/2020, and allows the holder to obtain £1 upon acquisition, and then either pay back £2 by 00:00:00 on 01/02/2020, or pay back £3 after 00:00:00 on 01/02/2020 and before 00:00:00 on 01/03/2020. The holder *must* pay one of these amounts by 00:00:00 on 01/03/2020. This contract is defined as follows:

```
truncate(<01/01/2020 00:00:00>, and(  
    one(GBP),  
    anytime(then(  
        truncate(<01/02/2020 00:00:00>, give(scale(2, one(GBP)))),  
        truncate(<01/03/2020 00:00:00>, give(scale(3, one(GBP))))  
    ))  
))
```

The first **truncate** combinator causes the contract to expire at the given date. The **and** combinator acquires both sub-contracts when acquired, i.e. **one**(GBP) and **anytime**(...). **one**(GBP) requires the counter-party to pay the holder £1 on acquisition. The **anytime** combinator allows the holder to choose when to acquire the sub-contract once acquired, but the sub-contract must be acquired before its horizon is passed and it expires. The **then** combinator acquires its first sub-contract if it hasn't expired, otherwise it acquires the second sub-combinator. If the **anytime** sub-contract is acquired before 00:00:00 on 01/02/2020, then the **then** combinator acquires the first sub-contract, which requires the holder to pay the counter-party £2 immediately due to **give** and **scale**. If the **anytime** sub-contract is acquired after this point, then the second **then** sub-contract is acquired, requiring the holder to pay the counter-party £3 immediately. The **anytime** sub-contract must be acquired before the **then** combinator expires, at 00:00:00 on 01/03/2020.

2.3.4 Representing Financial Contracts

In order to correctly design the full set of combinators for SmartFin, it is important to consider how SmartFin financial contracts will be represented in a smart contract implementation.

One way of representing these SmartFin contracts in a smart contract implementation is to use a functional approach. This involves creating a function representing each combinator, and having the functions carry out the actions which would occur when their respective combinator is acquired. This follows the description of the original DSL quite closely; however, due to the inherent statefulness required in these financial contracts (including state like time of acquisition, values of observables, etc), a functional approach is very difficult to implement. This is especially true as the state must be stored on the blockchain over a period of time between interactions with the smart contract. While a functional approach is possible, it would require workarounds to store combinators' state persistently - thus suggesting that this approach is not the best method of implementation.

SmartFin contracts can also be represented as structs where each combinator is represented by an object. For example, in the original DSL the contract defined as **scale**(5, **one**()) would be represented by an object representing **scale** which acquires its sub-object with 5 times the value, containing an object representing **one** which causes the counter-party to pay the holder 1 Ether at the time of acquisition. By using objects instead of functions, the issue of storing the state of each combinator can be solved trivially while still maintaining the compositional nature of combinators.

2.3.5 Difficult Combinators

Certain combinators of the original DSL described in 2.3.2 are difficult to represent directly in a smart contract environment. The **get** combinator acquires the sub-contract at its horizon, i.e. as late as possible before expiration. This is difficult to represent as callbacks cannot be scheduled in Ethereum smart contracts due to the necessity of deterministic evaluation on all nodes in the network, and thus a sub-contract cannot be automatically acquired at a given time.

Another problematic combinator is the **anytime** combinator. When acquired, this combinator requires the holder to acquire its sub-contract before the sub-contract expires. This is difficult to represent as there is no way to force the holder to acquire a contract (i.e. call an acquire function) in a given time-frame, or ever, from a smart contract. These issues will require further consideration when implementing SmartFin's smart contract implementation, and may require some compromises to be made.

2.3.6 Payment

Scheduled Payments

As described previously, time-based callbacks cannot be registered directly within an Ethereum smart contract to ensure determinism. This means that a smart contract cannot pay a user automatically at a given time, and so a financial contract which states that party A must pay party B at a given time is not able to be represented directly as a smart contract. It is possible to call a function on a smart contract at a given time through some external tool, but this effectively requires a trusted third-party (in the form of whoever is executing this tool). This is an issue which must be considered when implementing SmartFin’s smart contract behaviour.

Exhaustion of Funds

In the real world, if a financial contract exists between two parties, then any party which fails to fulfil their responsibilities as laid out in the contract can be sued for reparations. In the world of smart contracts, there is no such guarantee. If party A acquires a smart contract which says that party B will pay them some sum of money, party B must provide that money to the smart contract before payment can occur. If a smart contract does not have enough funds to fulfil all payments, there is nothing that party A can do to mitigate this issue (unless the smart contract is backed by some real-world legally binding contract). This also requires compromises to be made when implementing SmartFin’s smart contract behaviour.

2.3.7 Handling Observables

The original DSL can make use of external data in the form of *observables*. This allows a financial contract’s value to be modified based on external numerical data.

Defining Observables

The smart contract implementation of SmartFin needs some way of defining observables’ values. This is needed because the smart contract needs to know the values of each observable in order to calculate the value of payments, and so it needs to allow some way of obtaining/setting the value of each observable.

It could be useful to allow some other constraints to be set on observables; for example, if a financial smart contract pays out some value depending on an unbounded observable, there will be no guarantee on how much Ether must be provided for all payments to be fulfilled, and evaluation of the underlying SmartFin contract may not give a concrete value. This suggests that providing a bound for some observables may be useful for both guaranteeing payment and making evaluation of the SmartFin contract more useful.

Passing External Data to the Contract

Unfortunately, it is not possible for smart contracts to read data directly from an external server. This is because it could introduce non-deterministic execution to the EVM (i.e. two nodes executing the contract in different places could receive different data from the same address). This means that observables cannot be dynamically updated in the smart contract implementation of SmartFin.

Instead of dynamically reading data from an external source, observable values can be passed in through a function call. This can be done by defining a trusted source in the SmartFin contract definition, or by requiring the holder and counter-party to agree on a value (this more closely matches the original definition of an observable[27]); the latter system could be implemented through a signature-based system, or by storing suggested observable values of each party.

2.4 Development Tools

In order to implement a smart contract representation of SmartFin’s behaviour, a tool-set is needed for development and testing. There are several different toolchains which could be used for this, which are discussed and evaluated in this section. As smart contracts can be used to deal with real funds, any errors could potentially have a large real-world cost; as such, a testing framework will be a necessity to provide some basic evidence of correctness.

Local Ethereum Network

Ethereum allows developers to deploy their own local Ethereum blockchains, for testing purposes. These blockchains act like the default Ethereum blockchain, although they have several parameters which can be customised. Smart contracts can be hosted on these blockchains for testing purposes.

Parameters of the Ethereum network which can be modified include the difficulty of mining a block, the maximum limit a smart contract can cost to execute (in gas), the minimum gas a miner will accept, and a set of accounts to have Ether pre-allocated to them upon the creation of the blockchain. This allows the writing of arbitrarily complex contracts, the mining of blocks quickly and cheaply, and the ability to obtain Ether immediately - all of which are useful for testing contracts[13].

Setting up a bespoke blockchain just for testing smart contracts is quite complicated. There are many options for which values must be chosen, and not a huge number of resources which explain which option does what. This also only really provides the ability to manually test smart contracts through an Ethereum client, and does not provide any resources for automated testing (although external tools to do this do exist). It is still a useful option where public blockchains are not appropriate, however, and several clients allow easier set-up.

Truffle Toolchain

Another way that smart contracts can be tested is by using the Truffle toolchain. Truffle provides tools for running a local blockchain, smart contract compilation/deployment, automated testing, console interaction with contracts, build pipelines, and more[7]. The main tool that is needed for this project is an automated testing framework, which Truffle provides.

By using Truffle for automated testing, some continuous-integration can be put into place on the DSL. This enables rapid iteration of different smart contract functionality with lower risk of breaking existing features or introducing bugs. For these reasons (as well as ease-of-use), Truffle is a good option for the project’s development toolchain.

Remix

Remix is an online IDE for writing smart contracts in Solidity. It provides tools for writing, compiling, testing, debugging, and deploying smart contracts[14]. Smart contracts written in Remix can be deployed to a blockchain of the user’s choosing (either public or local). Function calls can then be made to the contract from the IDE, including payable functions (if authorised by some Ethereum account manager like MetaMask). It is also possible to define unit tests and run them from Remix, or from the command line[15]. This IDE can be a useful tool if any part of development requires the creation of smart contracts in Solidity.

Parity Tools

Parity Technologies provide a set of tools for developing smart contracts in Rust which compile to WebAssembly (*wasm*), including a set of Rust modules for interacting with Ethereum[44]. These

modules include:

- **pwasm-ethereum**: A module with bindings for interacting with an Ethereum blockchain, implementing functions like **sender** to obtain the sender of a function call transaction, and **call** to create and send a transaction[40].
- **pwasm-std**: A module implementing some features of the Rust standard library, which is incompatible with the **pwasm** modules[41].
- **pwasm-abi**: A module which provides macros for deriving a smart contract ABI (the smart-contract equivalent of an API) from a Rust trait, with annotations like **payable**, which indicates that a call transaction for the described function may contain Ether[39].
- **pwasm-test**: A module which provides mocking functionality for **pwasm-ethereum** function calls, to be used with automated testing[38].

They also provide the Parity blockchain client, which is a configurable client that can run a private blockchain capable of hosting wasm smart contracts (which not all blockchain clients are able to do)[46]. These tools make it possible to implement a smart contract representation of SmartFin’s behaviour in Rust, which may be a good option due to the relatively thorough documentation available for Rust in comparison to many smart contract languages.

Web3.js

Web3.js is a JavaScript package containing a collection of libraries for interacting with the Ethereum API[16]. This can be useful for deploying and interacting with smart contracts, and implementing integration tests (in conjunction with a testing framework). This package is also a useful option for implementing the external tool which deploys/interacts with financial smart contracts.

MetaMask

MetaMask is a browser extension which allows the user to authorise Ethereum transactions on a blockchain (either public or private) securely[31]. This enables manual testing of smart contracts, which is a valuable tool when testing Ethereum smart contracts written in any language. MetaMask is also a good option for connecting to a desired blockchain, and for deploying and interacting with financial smart contracts, from the external tool.

2.5 SmartFin Smart Contract Implementation

There are numerous design choices to be made with regards to the smart contract implementation of SmartFin. An important design choice is the form that SmartFin is implemented in; there are several major options, including:

- Creating a SmartFin library for an existing smart contract language, which can be called from smart contracts written by anyone.
- Writing a smart contract which implements SmartFin’s behaviour, and passing it a SmartFin contract definition.
- Creating a bespoke compiler for SmartFin which can compile SmartFin financial contracts to smart contracts.

There are several pros and cons to each of these choices, which are analysed in this section before a conclusion is reached on which method is most appropriate.

2.5.1 Creating a Library for an Existing Smart Contract Language

One way to implement SmartFin is to write a package for an existing smart contract language (like Solidity), and allow the user to call the required functions. With this method, the user can write a smart contract which calls into the relevant library functions for their SmartFin contract.

Solidity Libraries

Solidity, a high-level smart contract language for Ethereum, allows the writing and deployment of special smart contracts called *libraries*. Libraries are smart contracts which are deployed only once, and define functions which will be run in the calling smart contract's context[12]. This allows developers to define reusable functions in a single deployed library, and call said functions from any smart contract on the same blockchain.

Using Solidity's library functionality, it is possible to implement SmartFin's combinators as functions or structs in a library, thus allowing developers to access them from any smart contract they write - as long as the library is committed to the same blockchain.

This approach does not require implementing any compilation step to obtain a smart contract, making implementation more straightforward. The smart contracts which use a library would also be smaller in size compared to a smart contract which takes a SmartFin contract in the constructor, which would need to include all of the combinators' logic. This would reduce the amount of gas (Ether spent on transaction fees) required to deploy financial smart contracts.

One requirement of this approach is that the user must write a smart contract which calls these library functions. Unfortunately, this is problematic for a few reasons; firstly, these smart contracts can have code added which modifies the behaviour of the smart contract separately from the SmartFin contract. For example, before calling functions from the SmartFin library, a smart contract could immediately pay 1 Ether to the address of the counter-party. While this could be useful in specific situations, it clearly changes the value of the smart contract in a way unrelated to the SmartFin library calls, thus making it much more difficult to analyse. This level of freedom increases the risk of making unintended payments, whereas other methods of implementing SmartFin where a financial smart contract is output from a compiler or passed a SmartFin contract would not be affected by this problem, as the developer would not be expected to modify the smart contract directly.

Another problem with this solution is that it requires developers to understand both SmartFin and Solidity in order to create financial smart contracts. This adds a barrier to entry, and also increases the possibility of introducing errors which unintentionally affect the value of the financial smart contract.

A third issue is that analysing the value of a smart contract which calls functions from a library is difficult, as it requires analysing the smart contract to extract the definition of the SmartFin contract from within. This could be solved by requiring the developer to define the SmartFin contract inside the smart contract, and separately pass it into the analysis tool. This is not ideal as it may increase the risk of errors occurring; for example a developer may update their smart contract code but not the SmartFin contract passed into the analysis tool, thus receiving an inaccurate evaluation of the SmartFin contract.

EthPM - The Ethereum Package Management System

Another method of defining a set of structs and functions for representing SmartFin in Ethereum smart contracts is the use of EthPM, a package management system built for Ethereum. EthPM packages can include smart contract code, and the addresses of deployed smart contracts[6]. The

SmartFin DSL can be implemented as a set of functions or structs which are included in the EthPM package, and accessed from smart contracts which depend on it. EthPM would then handle the building of the final financial smart contract.

This unfortunately suffers from many of the same issues as writing a typical Solidity library. The user is still required to write a smart contract which calls the SmartFin functions, potentially introducing erroneous behaviour while doing so. The benefit of simpler redistribution of the package is also rendered slightly moot, as external tools for interacting with or analysing the value of a SmartFin contract cannot be distributed in an EthPM package, so another route of distribution will be needed either way. The added value of a package manager is also somewhat futile, as financial smart contracts should ideally have no other functionality besides the SmartFin function calls and thus shouldn't need to import any other packages.

2.5.2 Passing a SmartFin Contract to a Smart Contract

Instead of allowing the user to write their own smart contract code which calls into some SmartFin module, another option is to allow the user to write their SmartFin contract and pass it to a pre-written smart contract which implements the behaviour for any SmartFin contract definition. For this approach, the smart contract implementing SmartFin's behaviour can be written in any language which can compile to a smart contract.

One benefit of this method is that the end user only needs to implement a SmartFin contract, and *no* smart contract implementation. This lowers the barrier to entry for creating financial smart contracts, as the user only needs to learn how to use SmartFin, as opposed to an entire smart contract language like Solidity. This method also means that obtaining the SmartFin contract for analysis/evaluation is simple, as this contract can be defined separately from the smart contract implementation.

Additionally, passing the SmartFin contract to a pre-written smart contract prevents the user from modifying the final value of the smart contract by writing smart contract code outside of the SmartFin contract definition, as the user only writes their SmartFin contract and never touches the smart contract. Another benefit is that very little implementation work needs to be done from a compilation point of view, as the existing compilers for the high-level language used to implement SmartFin should handle the final compilation to a smart contract.

Implementing with a Traditional Smart Contract Language

Similarly to the implementation of SmartFin's behaviour as a Solidity library discussed earlier, the smart contract that implements SmartFin's behaviour can be written in Solidity (or another smart contract language). This smart contract can take and parse the SmartFin contract definition in the constructor, and then set its state so that its behaviour matches that of the given SmartFin contract.

Implementing with a Traditional High-Level Language

Instead of implementing SmartFin's behaviour in Solidity, it can also be implemented in some language which compiles to WebAssembly[19]; WebAssembly is typically more efficient than Solidity for numerous reasons - like the use of 256 bit integers in Solidity[11] - and it is being supported by W3C, a group consisting of multiple member corporations including Google and Mozilla[47], whereas Solidity is only supported by the Ethereum foundation. The ability to use tried-and-tested high-level languages would also be helpful, as they typically have more plentiful documentation and a more polished approach to language design compared to the relatively-new smart contract languages.

The main issue with this method is that producing a WebAssembly smart contract from a high-level language is not necessarily straightforward, as some functionality from the high-level language is often not usable on the Ethereum platform. This can make the implementation of SmartFin’s behaviour more difficult. The main features absent in ewasm are related to non-determinism[19], which shouldn’t cause many issues with the implementation of SmartFin’s behaviour which would not be present in all smart contract languages, so this should not be a major problem with regards to this specific method of implementation.

There are a few high-level languages which can be compiled to WebAssembly, but most of them are not viable for this project. Haskell has numerous ongoing WebAssembly compilation tools in development, but there are none which are implemented beyond the alpha stage[9][26]. Scala is in a similar situation, with an underdeveloped wasm tool-set[34]. Kotlin, Lua, Python, and TypeScript can all compile to WebAssembly, but do not provide tools for implementing Ethereum functionality like sending and receiving transactions[1].

Rust[37], on the other hand, has a set of tools released by Parity Technologies for compiling Rust programs to ewasm smart contracts[42], as previously described in section 2.4. These tools include bindings for Rust programs to interact with an Ethereum blockchain, alternative implementations of standard-library features which are incompatible with ewasm, tools for compiling smart contracts from Rust, tools for testing Rust smart contracts, and even a configurable blockchain client which can run these contracts. As such, Rust is a suitable high-level language to use to implement the smart contract which can represent any given SmartFin contract.

2.5.3 Creating a Bespoke Language and Compiling to a Low-Level Language

Instead of implementing a smart contract which represents the behaviour of any given SmartFin contract in a high-level language, SmartFin contracts could be compiled directly to low-level smart contract code (wasm, Yul[20], or similar) by implementing a bespoke compiler. Similarly to implementing SmartFin’s behaviour in a high-level language, this method also only requires users to define their SmartFin contract, with all of the same benefits.

The main practical change between this method and the method described in 2.5.2 is that SmartFin’s behaviour is implemented in a low-level language; this allows the implementation of SmartFin to use all of the available functionality, which may not be possible if compiling from a high-level language. This could allow improvements to SmartFin’s functionality which would not otherwise be possible.

The biggest problem with this method is that while implementing SmartFin in a low-level smart contract language may result in a more optimal implementation than the higher-level approach, it is much more difficult to write. Under the assumption that high-level languages have well-optimised compilers for producing low-level smart contract code, then the performance gains may not even be significant. Furthermore, if financial smart contracts are used within financial institutions then it is likely that the blockchain they are deployed to will not handle gas in the traditional manner, and would be more likely to have no gas payments, further reducing the benefits to improved performance. In these cases, implementing a compiler to compile SmartFin contracts to low-level smart contract code may only add to the effort required with little benefit.

2.5.4 SmartFin Implementation Summary

Overall, the problems with implementing SmartFin with a Solidity library directly affect the end user, making financial smart contracts more difficult to create, and making it easier to unintentionally alter the smart contract’s value. As such, a Solidity library/package is an unsuitable choice

for implementing SmartFin.

Implementing SmartFin’s behaviour in a high-level language or compiling from SmartFin to low-level smart contract code mitigates these two issues. Implementing SmartFin compilation *may* offer performance benefits, whereas high-level languages may be less difficult to work with; the performance benefits from low-level languages are also not guaranteed, and dubious in their utility. As such, the difficulty of implementing compilation to low-level contract code outweighs any expected benefits, and implementing in a high-level language which compiles to WebAssembly is the more suitable option.

2.6 External Tool for Financial Smart Contract Interaction

In order to compose and evaluate financial contracts written in SmartFin, and to deploy and interact with their smart contract representations, an external tool is needed. As discussed in section 2.4, there are several browser-based tools which facilitate interaction with Ethereum blockchains, namely MetaMask and Web3.js. Thanks to these tools, implementing a web client to handle these external functions is a viable solution.

While other languages provide tools for interacting with Ethereum blockchains (like the Python implementation of Web3[17]), the large swathe of support for blockchain interaction in the browser makes a web client a very attractive option. Package management tools like Yarn[8], graphical frameworks like React[24], and testing frameworks like Mocha[21] can also aid rapid development of the web client. Furthermore, MetaMask provides an easy to use and secure method of connecting to the blockchain, and it is only available as a browser extension[31]. As such, implementing the external tool as a web client is a valid solution which enables rapid development and has few obvious issues.

2.7 SmartFin Contract Cost Analysis

A useful feature provided through the external tool is the mathematical evaluation of a given SmartFin contract’s potential cost/value with any given acquisition times. This is particularly useful to prevent the writing of SmartFin contracts with unexpected behaviour. The mathematical implementation of such a tool for the original DSL is outlined in the paper by Peyton Jones et al.[27].

The value of each contract is dependent on its sub-contracts. For example, in the original DSL the contract `and(one(GBP), one(GBP))` will have a value of $(\mathcal{L}1 + \mathcal{L}1) = \mathcal{L}2$, thus the value of an `and` combinator is equal to the sum of its sub-contracts’ values. This makes the general approach behind contract evaluation fairly straightforward, where the contract is treated as a tree with the top-level contract as the root, and all sub-contracts as children. The tree can be traversed depth-first, and then backtracked to the root while accumulating the required values.

There are other evaluation issues that the paper by Peyton Jones et al.[27] deals with, such as inflation rates and exchange rates. Because this project is concerned with a smart contract implementation of SmartFin, financial smart contracts will only deal in the relevant cryptocurrency (Ether), and thus evaluation based on exchange rates of payments is slightly irrelevant. The use of Ether also makes the analysis of inflation quite difficult, as cryptocurrencies are notoriously volatile in value. These two features are therefore less useful (albeit not useless) and require significant work to implement; as such, they are not an essential feature for the analysis of SmartFin contracts.

One potential issue with the evaluation of SmartFin contracts is the presence of observables; in order to obtain a value of a SmartFin contract with a given acquisition time, the value of any observables must be evaluated. This is a difficult problem to solve, as modelling the value of

observables requires allowing users to input some representation of a numerical model representing this value over time, and the implementation of algorithms to evaluate these models. This is a huge undertaking, and as such a more basic evaluation mechanism which does not estimate the value of observables is a more viable option given the time constraints.

2.8 Remarks

After significant background research, certain conclusions can be reached with regards to the approaches taken for this project. In order to represent SmartFin contracts as financial smart contracts, Ethereum is the most viable smart contract platform due to its sizeable user-base, the vast set of development tools available, and the platform's user-friendly nature.

A smart contract representation of SmartFin can generally follow the intended financial contract behaviour accurately, but certain issues may arise. Scheduling of payments is effectively impossible due to the distributed and deterministic nature of Ethereum blockchains, where scheduled callbacks are impossible without requiring external interaction. Exhaustion of funds is also a problem which may occur with financial smart contracts, as there is no way for a smart contract to ensure that a user pays the amount required by the contract definition. Observable values will also be impossible to obtain dynamically due to the deterministic nature of the blockchain, and thus they will need to be passed to the contract in some way. These issues must all be dealt with when implementing a smart contract representation of SmartFin.

When developing the smart contract implementation of SmartFin, allowing the SmartFin contract definition to be passed through the constructor is a beneficial approach as it prevents the user from altering the smart contract's behaviour directly, and makes analysis of the SmartFin contract easy. Compiling directly from the SmartFin contract to a low-level smart contract is comparatively very difficult to implement for little gain, and requiring the user to write a smart contract which calls into a SmartFin library would make analysis difficult and risk the introduction of errors. Implementing the smart contract representation of SmartFin in Rust/wasm ensures long term support from the W3C group, and improved performance compared to Solidity and other smart contract languages - as such, it is the option of choice.

In order to implement the external tool for evaluating SmartFin contracts and deploying or interacting with financial smart contracts, a web client makes a lot of sense. This is because of the multitude of web-development tools available, and the existence of various modules/extensions for communicating between a blockchain and a web client. These factors ensure that development of the web client can be rapid and problem-free, and that the external tool can carry out all of the required functionality.

The following chapters will describe the design of the SmartFin DSL, the smart contract implementation of SmartFin and its combinators, and the web client used to interact with financial smart contracts.

Chapter 3

The Design of SmartFin

The main goal of this project is to provide a smart contract implementation of a DSL for financial contracts. Before discussing this implementation, this chapter defines the syntax and semantics of the DSL being implemented, SmartFin.

The syntax and semantics of SmartFin are derived from a DSL proposed by Peyton Jones et al.[27], described in section 2.3, but SmartFin varies slightly from this original DSL. The smart contract implementing SmartFin must behave correctly when any SmartFin contract definition is provided, according to the syntax and semantics laid out in this chapter.

3.1 Overview of SmartFin Financial Contracts

SmartFin is a combinator domain-specific language used to describe financial contracts, and in general operates very similarly to the DSL defined by Peyton Jones et al.[27] which is described in section 2.3.1. The general gist is that a SmartFin contract represents a financial contract between two parties, which can define payments between these parties. The *holder* party is able to acquire a SmartFin contract - just as a traditional financial contract would be signed - and any obligations are carried out subsequently. SmartFin contracts can expire, and contracts can rely on external input.

3.2 Syntax

The syntax of SmartFin is very simple, due to its combinatorial nature. The Backus-Naur form of the grammar is as follows, where `<name>` is a string starting with a non-integer character, `<time>` is a date and time, `<integer>` is an integer value, and `<address>` is an Ethereum address:

```
<contract> ::= 'zero'
              | 'one'
              | 'give' <contract>
              | 'get' <contract>
              | 'anytime' <contract>
              | 'scale' <observable> <contract>
              | 'truncate' <time> <contract>
              | 'and' <contract> <contract>
              | 'or' <contract> <contract>
              | 'then' <contract> <contract>

<observable> ::= <integer> | <name> <address>
```

This syntax has one major difference from the syntax of the original DSL presented in the paper by Peyton Jones et al. [27]. This difference is the representation of *observables*, used with the `scale` combinator in the `<contract>` rule. In the original DSL, an `observable` takes a constant value, or a representation of a time-varying value, but its syntax was left abstract due to the somewhat abstract nature of the combinators' implementation in the paper. The paper was more focused on evaluating financial contracts written in this form than representing them programmatically. As this project requires a concrete implementation of the DSL, the syntax cannot be left abstract. Because of this, it is necessary to give a specific definition of an observable's syntax; the chosen definition being a constant integer value, or a name and an Ethereum address. The purpose of these representations is explained further in section 3.3. The original DSL also allowed observables to have mathematical operations carried out on them, whereas SmartFin does not.

For clarity, the format of the `<time>` rule in the provided BNF can be either UNIX Epoch time, or a date and time in the format "`<DD/MM/YYYY HH:mm:ss +ZZ>`" - where "DD" is a two-digit date, "MM" is a two-digit month, "YYYY" is a four-digit year, "HH" is a two-digit hour, "mm" is a two-digit minute, "ss" is a two-digit second, and "+ZZ" is an optional positive or negative two-digit time zone offset.

3.3 Semantics

Each combinator has a specific meaning, potentially depending on sub-combinators, times, and/or observables. This section gives a definition of the semantics of SmartFin combinators where they differ from the original DSL, namely `one` and `scale`. For combinators not listed here, see 2.3.2 for their semantics. The conventions used to represent the syntax of these combinators is defined in figure 3.1, and the definition of each combinator's semantics is as follows:

<i>c, d</i>	<i>Contract</i>
<i>o</i>	<i>Observable</i>
<i>x</i>	<i>Constant Value</i>
<i>n</i>	<i>Observable Name (as in section 3.2)</i>
<i>a</i>	<i>Ethereum Address</i>
<i>t</i>	<i>Time</i>

Table 3.1: Conventions for the description of the SmartFin's semantics

`zero`, `give c`, `and c d`, `or c d`, `truncate t c`, `then c d`, `get c`, `anytime c`

These combinators' semantics are unchanged from the original DSL, described in section 2.3.2.

`one`

This combinator represents a contract which requires the counter-party to pay the holder 1 Wei upon acquisition. The currency of payments is always Ether, as opposed to the original DSL's `one` combinator which takes a currency. This contract can be acquired at any time, and thus has no horizon.

`scale o c`

A `scale` combinator represents `c` with all payments multiplied by the value of the observable `o` at the time of acquisition. `scale(o, c)` has the same horizon as `c`.

An observable can be provided in the format `x`, or `n a`. An observable in the form `x` is a constant integer value. For example, `scale 5 one` requires the counter-party to pay the holder 5 Wei on

acquisition.

An observable in the form $\mathbf{n} \ \mathbf{a}$ is a time-varying value. The name \mathbf{n} is an identifier for the observable, and the address \mathbf{a} represents an arbiter for the observable's value. This address is treated as the canonical source of the observable's value at the time of acquisition. No two observables may have the same name and arbiter, or there would be no way to differentiate the two, even though their values may differ (if they are acquired at different times, for instance).

3.4 Remarks

Overall, the SmartFin DSL is very similar to the original DSL described by Peyton Jones et al.[\[27\]](#), with a few differences for the sake of compatibility with the Ethereum platform. The next chapter will describe the top-level implementation of a smart contract which can represent any given SmartFin contract, and the following chapter describes the implementation of specific combinators' semantics.

Chapter 4

Smart Contract Design and Implementation

One of the major parts of this project's contributions is a smart contract with the necessary functionality to represent any given SmartFin financial contract definition (passed into the constructor), which is mainly interacted with through a web client. The syntax and semantics of SmartFin are described in chapter 3. This chapter will detail the design and implementation of this smart contract's general operation and ABI (the smart contract equivalent of an API), and evaluate to what level it is sufficient for its purpose of representing SmartFin contracts.

The elements covered in this chapter are displayed in a dependency graph in figure 4.1 . This chapter does not go into depth on the combinator implementation modules, which are instead described in chapter 5.

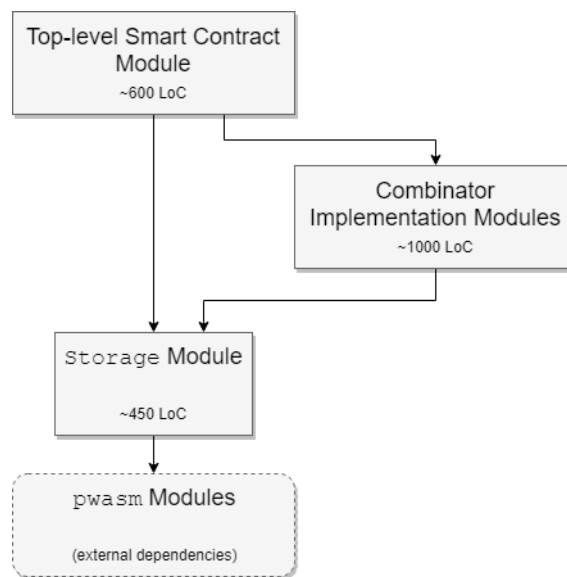


Figure 4.1: A dependency graph of the modules used for the SmartFin smart contract, with their approximate *Lines of Code* (not including tests). `pwasm` modules are external dependencies, described in section 4.3.1. The combinator implementation modules are described in chapter 5.

4.1 Requirements

In order to ensure that the smart contract accurately and correctly represents SmartFin contracts, a set of requirements has been devised. These requirements form the motivation behind specific design and implementation choices made during the creation of the smart contract.

The SmartFin contracts that can be represented by the smart contract consist of an agreement between two parties (the *holder* and *counter-party*) with regards to payments between these parties. As such, the smart contract must hold the definitions of both the holder and counter-party, and allow the two parties to exchange funds through the contract.

The holder should have the ability to acquire the contract (just as they would be able to sign a traditional financial contract), and this should result in the evaluation of payments to be made between the two parties with respect to the acquisition time.

The smart contract should also be able to represent the behaviour/state of the combinators defined in section 2.3. The evaluation of the payments between the two parties should respect the semantics of these combinators.

Certain values in the SmartFin contract may be unknown until a later point (for example, observables in a `scale` combinator), and so the smart contract requires some way of obtaining these values. The holder may also be able to choose certain options on the SmartFin contract (for example, choosing a branch of an `or` combinator), and the smart contract should also facilitate this.

The evaluation of payments between the two parties may change over time. For example, the SmartFin contract `get truncate (01/01/2020 12:00) one` requires a payment to occur from the counter-party to the holder at noon on the 1st of January 2020. Before this time, no payments should occur. This means that the smart contract must be able to keep track of the state of the SmartFin contract's obligations over time.

All of these are specific requirements that are required for the smart contract to accurately represent any given SmartFin contract. These have been taken into consideration during the design and implementation of the smart contract, informing the final product.

4.2 Design

The overall design of the smart contract is fairly simple, albeit with several caveats. From a very high level, the smart contract consists of a class with an ABI, several state variables, and a tree of `ContractCombinator` objects. The behaviour of the SmartFin combinators is implemented by the `ContractCombinator` objects, the entire tree represent the SmartFin contract definition, and the rest of the smart contract effectively informs the way that it operates as a whole. See section 5.1 for details on the representation of individual SmartFin combinators.

4.2.1 High Level Design Challenges

In an ideal world, the smart contract would track payments over time and exchange funds between the required parties immediately, as one might expect from a web-service or similar. Unfortunately, the smart contract runs on a blockchain, and as such there are several restrictions which affect its ability to operate as required. These inform the high-level design of the smart contract.

Payment Evaluation

It is not possible to schedule callbacks on a blockchain without relying on external tools, as this would jeopardise determinism, and it would require a miner to run the code at a specific time. As such, the smart contract is only able to perform execution when called into from a user.

Certain SmartFin contracts require payments to occur at specific times; for example, the contract `get truncate t one` requires the counter-party to pay the holder 1 Wei at time t . Unfortunately, smart contracts are unable to make payments at specific times, as this would require scheduled callbacks to be possible.

In order to represent time-respective payments, the implemented smart contract allows payments to be fulfilled retroactively. In essence, the smart contract does not perform payments at specific times, but instead keeps track of important times - typically combinators' acquisition times - and evaluates which payments should have occurred since that time after the fact.

To facilitate this, payments between the two parties are represented as a balance, changing when either party stakes/withdraws funds to/from the contract, or when payments are evaluated. In order to evaluate the payments which should have occurred by the current point in time, the smart contract has an `update` method which calculates the new balance between the holder and counter-party, resulting from any payments defined by the SmartFin contract definition between the last `update` call and the current block-time. For example, when the holder acquires the contract `get truncate t one` before time t , their balance does not change, and their balance does not change when t passes either. Only once the contract is updated *after* time t does this balance change. This is depicted in figure 4.2.

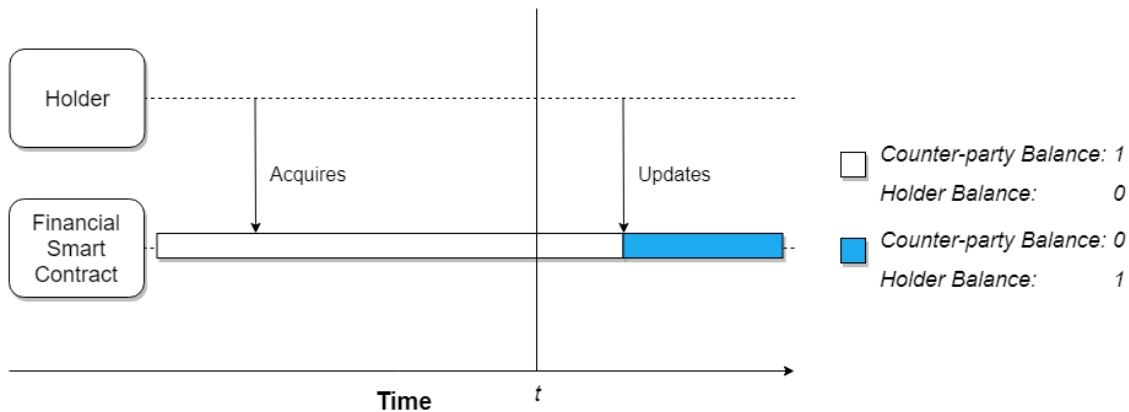


Figure 4.2: A diagram depicting the balance of the holder and counter-party over time for the financial smart contract representing the SmartFin contract `get truncate t one`.

Observable Evaluation

Similar to the evaluation of payments, another factor in SmartFin contracts which changes over time is observables. These are objective, potentially time-varying quantities. An example of an observable which does not vary over time would be the number 5, whereas an example of an observable which does vary over time is the average temperature of London in Celsius. Both are objective, but one takes different values depending on what time you sample it.

Unfortunately, this poses a couple of problems for the smart contract. As mentioned earlier, smart contracts cannot schedule callbacks at specific times, and so finding the value of an observable at a

specific time is not a simple task. There is also the issue that smart contracts are unable to access information which is not stored on the blockchain, which means that sampling observable values from websites or other off-chain sources is not possible.

These two restrictions effectively rule-out the possibility of the smart contract obtaining the value of a time-varying observable without external interaction from the user. To work around this issue, when a SmartFin contract relies on an observable the author must define an *arbiter* for its value. This arbiter is an address which can provide the observable's acquisition-time-value to the contract, effectively acting as a trusted source of information.

When an observable's value is unknown while the smart contract is being updated, the sub-contract relying on the observable will not be updated until the value is provided. After its provision, the sub-contract can be updated, thus allowing retroactive provision of observable values. For example, take the financial smart contract representing the SmartFin contract `scale x <arbiter address> one`; when acquiring the contract before the observable's value is set, the balance does not change. Once the arbiter sets the value of the observable to 1, and then the financial smart contract is updated, the balance changes. This is depicted in figure 4.3.

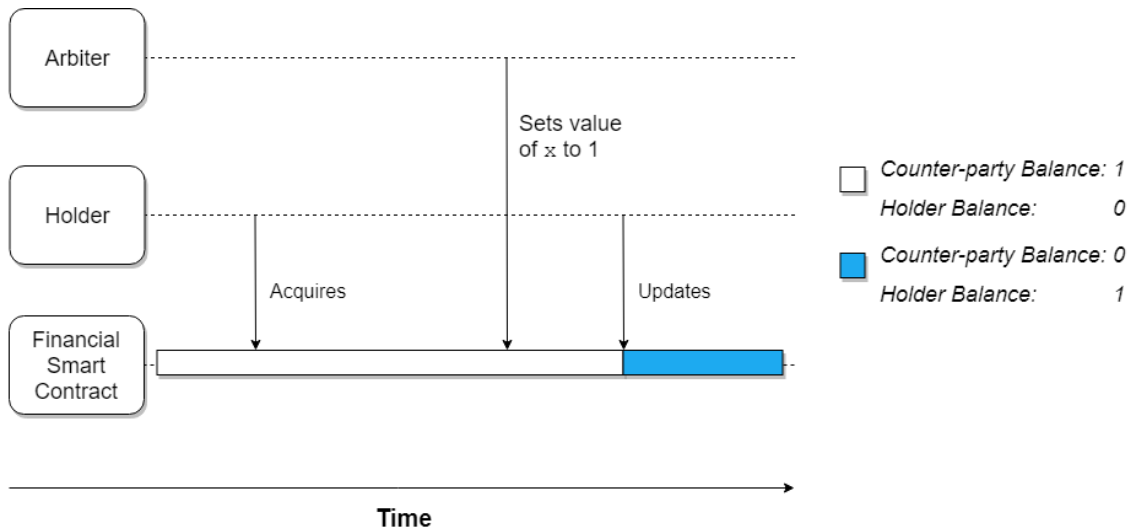


Figure 4.3: A diagram depicting the balance of the holder and counter-party over time for the financial smart contract representing the SmartFin contract `scale <observable name> <arbiter address> one`.

Exchanging of Funds

As you have hopefully noticed by now, smart contracts deal heavily with the exchanging of funds. Regular financial contracts are typically binding, requiring one party to pay the other party or risk legal action. Smart contracts are a little different; they have no way to ensure that one user will pay another user, but instead act as a separate entity which processes payments from both both parties.

A disadvantage of this is that the implemented smart contract is unable to force one party to pay the other party. It can maintain a balance between the two parties, indicating when one party *should* pay the other, but it has no means to enforce this payment. In order to somewhat circumvent this issue, the smart contract allows users to stake Ether in the contract at any time (increasing their balance), and to withdraw Ether under certain conditions. In order to withdraw an amount of Ether:

- The user must have enough Ether in their balance.
- The smart contract must have enough total balance.
- The user's balance must be larger than withdrawal gas fees (where gas fees apply).

By allowing the parties to stake Ether in the smart contract beforehand, the smart contract is able to make any required payments without immediate action from the paying party as long as it has enough funds. Incidentally, this is another reason that retroactive payments are appropriate, as there is no guarantee that the smart contract will actually have the funds to pay a party when a payment should occur.

Unfortunately, it is still impossible for the smart contract to solve the issue of binding a party to payment. As such, if one party decides not to pay, they can simply refuse to stake any Ether and thus the smart contract will have no funds to pay the other party. There were a few possible solutions (or at least mitigations) to this issue.

Restricting withdrawal from the smart contract until its conclusion would prevent parties from withdrawing before they have to make a payment. Unfortunately, it would also lower the value obtained through the contract, as funds locked in the contract's balance could instead be earning money through investment. Another potential solution could be to prevent contract acquisition until sufficient funds have been staked in the smart contract, but this has a similar issue that funds staked in a smart contract would be better served elsewhere.

The best solution to this issue is to use financial smart contracts alongside a legally-binding contract requiring a party to stake funds when required. For now, we assume that each party is bound to make payments to the smart contract in an honest way, as it seems that this is not something that can be solved in a satisfactory manner through the smart contract itself.

4.2.2 Financial Smart Contract ABI

The smart contract has a set of functions which can be called by the web client, allowing it to interact with and monitor the contract. This ABI is informed by the requirements of the smart contract, and any caveats encountered due to its nature as a smart contract. The behaviour of many of these functions is determined by the composition of combinators used to write the SmartFin contract passed-in to the smart contract's constructor. The full financial smart contract ABI is listed in [appendix A](#).

Constructor

The constructor of the smart contract is responsible for initialising the smart contract object, and as such it requires all information relevant to the initial state of the SmartFin contract. The constructor takes the following parameters:

- The SmartFin contract definition.
- The address of the contract holder.
- Whether or not the contract should allocate gas fees upon withdrawal (as private blockchains do not always require gas fees).

Pure Functions

The smart contract defines several functions which return the value of state variables (i.e. *getter* functions). This enables the web client to display the current state of the contract. As these are *pure* functions, they do not cost any gas to call, and thus the web client may update its view of

the smart contract as often as required at no cost. These functions consist of the following:

- `get_holder`: Returns the address of the contract's holder.
- `get_counter_party`: Returns the address of the contract's counter-party.
- `get_contract_definition`: Returns the SmartFin contract definition.
- `get_balance`: Returns the balance of one of the parties (in Wei).
- `get_concluded`: Returns whether or not the smart contract has concluded execution (i.e. no more updates are required).
- `get_use_gas`: Returns whether or not the smart contract allocates gas fees upon withdrawal.
- `get_last_updated`: Returns the block-time that the last call to the `update` function occurred at.
- `get_acquisition_times`: Returns a list of the acquisition times of the contract and its sub-contracts.
- `get_or_choices`: Returns a list of choices which have been made for `or` combinators.
- `get_obs_entries`: Returns a set of observables, including their names, values, and arbiter addresses.

Setting Values

The smart contract has several values which rely on user input to evaluate. One such combinator is the `or` combinator, which allows the holder to acquire either of its two sub-contracts at the time of acquisition. The holder is able to call the function `set_or_choice` at any time, to inform the smart contract of which sub-contract the holder should acquire for a given `or` combinator. An `or` choice cannot be changed after it has been made, to prevent inconsistency across multiple updates.

Another combinator which requires user input is the `scale` combinator. For each `scale` combinator's observable, if said observable is not a constant then it will require an arbiter to provide its value. The arbiter can call the `set_obs_value` function to set the value of an observable at any time, once per observable.

Acquisition

The holder must have the ability to acquire the smart contract, just as a prospective holder would be able to acquire a traditional financial contract by signing it. As such, the smart contract's ABI contains an `acquire` method, callable by the holder. This method updates the contract state to store the time of acquisition.

Besides the top-level contract, `anytime` combinators can also be acquired at certain times during the smart contract's lifetime. As such, there is also an `acquire_anytime_sub_contract` method; if the acquisition is valid (i.e. the `anytime` combinator's parent is acquired, and the combinator is not expired), then the smart contract will update its state to reflect that the sub-contract has been acquired.

Update

Due to the nature of the blockchain, the smart contract cannot keep itself up-to-date without external action. The `update` function causes the smart contract to evaluate the difference in balance between the current `update` call and the last `update` call, retroactively evaluating payments. The `update` function can be called from any address, as the update operation is handled entirely by the smart contract with no external inputs.

Payment and Withdrawal

The smart contract allows either party to stake Ether, to facilitate payments between them. The `stake` method is a *payable* function (i.e. it can receive Ether), which increases the balance of the party that called it by the amount of Ether received.

The smart contract also allows either party to withdraw Ether, assuming the balances allow it. To do this, the `withdraw` method takes an amount and sends the requested amount of Ether to the calling party, reducing their balance by the withdrawn amount. Gas fees (when enabled) are taken out of the party's balance on top of the withdrawal, or out of the withdrawal amount if the balance hits 0. Besides the two involved parties, no addresses can stake or withdraw Ether - it would be unclear who to allocate staked Ether to, and only the respective party should be able to withdraw from their balance.

4.3 Implementation

4.3.1 Technologies

In order to implement the smart contract, there were several choices to make with regards to which technologies would be employed. As discussed in section 2.5, there are many possible tools with which to implement a smart contract.

The final method that was decided upon for implementing the smart contract was the use of Rust[37] with *pwasm* modules/tools provided by Parity Technologies[44] for compilation to WebAssembly. The final WebAssembly contract is compatible with private blockchains run from Parity's own Ethereum client, as well as with the Kovan test network and other wasm-compatible Ethereum clients[43].

The *pwasm* modules provide various tools for developing smart contracts in Rust, as discussed in section 2.4. The main tools which were made use of in the smart contract implementation were:

- ***pwasm-std***: A module re-implementing useful *pwasm*-compatible functionality from the Rust standard library (which is not compatible with these *pwasm* modules).
- ***pwasm-ethereum***: A module providing access to information and functionality from the blockchain, including the current block-time, the address of the caller, writing to or reading from smart contract storage, and more.
- ***pwasm-abi*** and ***pwasm-abi-derive***: Modules which enable the conversion of a Rust trait into a smart contract ABI definition.
- ***pwasm-test***: A unit testing framework for smart contracts written in Rust.

Unfortunately, due to the *pwasm* modules' incompatibility with the Rust standard library, it was impossible to use most external modules for the Rust smart contract. The smart contract implementation was possible without requiring any more external modules, but certain problems like serialization of structs could be solved much more easily using external libraries if it were possible.

In order to build the Rust smart contract, *Cargo*[36] (the Rust package manager) was used. Parity also provides a build tool for building wasm contracts with *Cargo*, *wasm-build*[45]. *Web3.js*[16] was also used to deploy smart contracts to the blockchain, and interact with them (for testing purposes). This was managed with *yarn*[8], a package manager for JavaScript.

4.3.2 Smart Contracts in Rust

Rust smart contracts implemented using the pwasm modules must be defined in a particular manner. Firstly, the Rust program must have a few things implemented in the main module of the application (typically written in `lib.rs` for a Rust program), including a smart contract ABI, a smart contract struct, and a `call` and `deploy` function.

The smart contract's ABI is defined through a trait (a set of several methods) with annotations. Pure methods (i.e. those that do not alter smart contract state) are denoted by the `constant` annotation, and payable methods (i.e. those that can receive Ether) have the `payable` annotation. There must also be a `constructor` method implemented, which initialises the smart contract struct/state on deployment. This trait is then used to generate the smart contract ABI, as well as to generate an *endpoint* which delegates method calls to the smart contract struct. Only certain types are able to be sent/received via a pwasm smart contract's ABI, including integer types, vectors, addresses, booleans, and a few more binary data types.

A smart contract struct must also be implemented. This is a data type which must implement the ABI trait. Every time the smart contract is called into, a new instance of the smart contract struct is constructed. This means that member variables on the struct will be reset every time that the smart contract is called into. In order to store persistent state across smart contract calls, the smart contract storage must be manually written to/read from (using methods provided by `pwasm-ethereum`).

Alongside the ABI and smart contract struct, public static `call` and `deploy` functions must be implemented. The `call` function will initialise the smart contract struct, and an *endpoint* which delegates method calls based on the ABI, and handles smart contract calls by providing the call inputs (method name, parameters, etc.) to the endpoint. The `deploy` function is similar, but is called when the smart contract is first deployed to forward the inputs for the constructor call to an initialised endpoint.

4.3.3 ABI Methods

Constructor

The constructor is called when the smart contract is initially deployed, and is responsible for setting up the initial state. Firstly, the validity of the constructor arguments is checked, throwing an error if any invalid arguments are passed; throwing an error will cause smart contract methods to cease execution and roll-back state changes. After checking the arguments, the constructor writes several values to storage, including the holder and counter-party addresses, the holder and counter-party balances, whether or not the contract uses gas, the last-updated time, and the contract definition.

The contract definition is passed to the constructor as the type `Vec<i64>`, i.e. a vector of signed 64-bit integers. This is necessary as Rust smart contracts written with pwasm modules cannot send/receive strings over the ABI, and thus the contract definition must be provided in a serialized form. The serialized form of SmartFin simply involves representing the SmartFin contract as an array of integers. Each combinator is represented by an integer (0 to 9), times are represented as UNIX Epoch times, integer values are unchanged, names are represented by arrays of character codes, and addresses are represented by an array of four integers. The contract definition is deserialized recursively into a set of `ContractCombinator` objects, representing the SmartFin combinators. For more information on how the combinators are represented in the Rust smart contract, see chapter 5.

Pure Methods

The pure ABI methods (i.e. getters) are generally quite straightforward; `get_holder`, `get_counter_party`, `get_contract_definition`, `get_balance`, `get_use_gas`, `get_last_updated`, and `get_acquisition_times` all read the required data from storage and return it in a trivial manner.

`get_concluded` returns `true` if the contract will no longer change state over time, i.e. all responsibilities are fulfilled. This is true if the contract has been acquired and the combinators have been fully updated to their final state, or if the contract has expired before acquisition.

`get_or_choices` reads the set of choices for `or` combinators from storage, and then serializes this into a vector of bytes. 1 represents the first sub-contract, 0 the second, and 2 means that no choice has been made yet.

`get_obs_entries` constructs and returns the set of observable entries for the contract from storage. This consists of a vector of observables, with their arbiter address, value, and name (all serialized as several signed 64-bit integers).

Sub-contract Acquisition and Setters

The functions setting `or` combinator choices, observable values, and acquiring `anytime` sub-contracts are also quite straightforward. The `or` choices, observable values, and `anytime` acquisition times are all stored as vectors in the smart contract storage. In order to update any of these values, the relevant element in the vector is written to directly through a storage object, described in section 4.3.4. Acquiring an `anytime` combinator's sub-contract will also update the contract, with a call to `update`.

Acquire

The `acquire` method first reads and deserializes a combinator object from storage, which implements a combinator contract's behaviour. If the caller is the holder and the combinator hasn't been acquired already, the method calls an `acquire` method on the contract combinators, passing down a reference to the storage object. For more information on the combinator objects' `acquire` methods, see section 5.2. The combinator object is then serialized and re-written into storage.

Update

The `update` method first checks if the contract has concluded. If so, then updating the contract will have no effect no matter what happens, so the contract throws an error. After this, the combinator object is read from storage and deserialized, as in the `acquire` method. The storage object is passed to the `update` method on the combinator object, which is also described in section 5.2. The combinators are then serialized and written back to storage.

After updating the combinators, the new balances of the holder and counter-party are written to storage. These are calculated using a `safe_add` method, which checks for integer overflow. Rust should throw an error automatically if it occurs, but extra precaution is taken due to the severity of overflow errors with balances in smart contracts.

Stake

The `stake` method is a `payable` function, to which Ether is paid upon calling. The `pwasm-ethereum` function to obtain the value of a transaction returns a `U256`, i.e. an unsigned 256-bit

integer. Balances are stored in the smart contract storage as signed 64-bit integers, as they are easier to work with being Rust primitive types. Because of this, the `stake` method must first check that the value provided can be converted into a signed 64-bit integer, or an error is thrown.

After checking this, the method checks whether the caller is the holder or the counter-party - an error is thrown otherwise. The method then safely adds the value of the method call transaction to the balance of the relevant party, and writes the new balance to storage.

Withdraw

The `withdraw` method allows a user to withdraw funds from the contract by initiating a transfer of Ether from the smart contract to the caller. First, the method checks which address to withdraw Ether from. If the caller is not one of the holder or the counter-party, then an error is thrown.

The method calculates how much Ether to send by comparing the amount requested to the balance of the relevant party, the total contract funds, and the gas cost. If the contract allocates gas fees upon withdrawals, then 2300 Wei in gas fees is added to the withdrawal amount. If either of the total funds of the contract or the party's balance are smaller than the gas cost, then an error is thrown. The withdrawal amount (including gas fees) is clamped to the minimum of the party's balance and the total funds. If the final amount is 0 or less, then an error is thrown.

After the viable withdrawal amount is calculated, the relevant balance is updated, and then the method creates a transaction from the smart contract to the caller address. If the transaction fails then the relevant balance is rolled back to its original amount, and an error is thrown.

4.3.4 Storage

Besides the ABI methods and combinator behaviours, the other main functionality that the smart contract implemented was storage. State - such as the holder address, acquisition times, or the combinator objects - must be stored in smart contract storage in order to remain persistently across multiple smart contract method calls. In order to do this, the `Storage` module was implemented to handle the storage of various data types.

Memory Layout of Smart Contract Storage

Smart contract storage is addressed using 256-bit integers (of the type `H256`), and each slot in storage can store 256 bytes of information in the form of an array of 32 unsigned 8-bit integers. In order to separate storage into different spaces, the most-significant byte of each memory address is treated as a "namespace" byte by the smart contract. The hexadecimal addresses `0x0000...00` to `0x00FF...FF` span the first memory namespace, `0x0100...00` to `0x01FF...FF` span the second memory namespace, etc. As such, if a function is writing to storage sequentially and reaches the end of a namespace, by incrementing the most-significant byte, then an error must be thrown.

Storage Module Design

The `Storage` module contains definitions for a `Storage` struct and several traits which include methods to read from/write to smart contract storage. The reason that a struct is used instead of static methods is that data read from/written to smart contract storage is cached in the `Storage` object, and so only one read to smart contract storage must occur for each object being read per ABI method call. The cache is destroyed between ABI method calls as the smart contract struct is destroyed after the ABI call finishes and the smart contract is exited. Cached reads/writes are stored in a vector of `Entry` struct instances, which contain the storage address and a value.

There are also three generic traits which are used for storage; the `StoresFixed<T>` trait indicates that the implementing struct can store a value of the type `T` in memory, and that the size of any values of this type is constant. The size restriction is helpful as it ensures that values can be stored sequentially in smart contract storage without changing in size and overwriting each other, or leaving empty space between elements. The trait defines methods for reading/writing a value, and a static `size` method.

The second generic trait used for storage is `StoresFixedVec<T>`, which indicates that the implementing struct can store an object of the type `Vec<T>` in memory, where elements of type `T` have a known fixed size. This trait allows reading/writing the whole vector to storage with linear time complexity, as well as getting the length, reading/writing individual elements, and pushing elements to the end of the array with constant time complexity.

The third trait used for storage is `StoresVariable<T>`, which indicates that the indicating struct can store an object of the type `T` in memory, but its size is unknown and/or variable. This trait allows writing or reading of whole objects of type `T` with linear time complexity.

All write and read methods return a key along with any other value. This key represents the last storage address used in the method. For example, if a vector is stored at address 2 and takes up 6 slots, the read/write methods will return address 7. This enables sequential writing of values in memory, especially useful when those values have variable size.

Storage Method Implementation

Besides several implementations of the traits mentioned above, the `Storage` struct also has method implementations for several of its own helper methods, for converting values between types. The `Storage` struct also has a constructor which will initialise the storage cached vector to empty.

Only one type can be read from/written to smart contract storage using the `pwasm-ethereum` `read/write` functions, an array of 32 unsigned 8-bit integers (i.e. `[u8; 32]`). Several methods on the `Storage` struct implement conversion of other types to/from this type. The `from_address` method converts an address into an `H256` data type using the `H256::from` function, which can be converted directly into a `[u8; 32]` object. The method `to_address` converts similarly in the opposite direction.

The `from_bool` method converts a boolean value into `[u8; 32]`. In the case of `false`, the function returns a `[u8; 32]` array with all elements set to 0, otherwise an element is set to 1. The `to_bool` method can thus return whether or not the given `[u8; 32]` is not equal to `H256::zero()`.

The last pair of conversion functions that the `Storage` struct implements is `to_i64` and `from_i64`. As the `pwasm-std` module implements functions `read_u64` and `write_u64`, which convert an unsigned 64-bit integer (i.e. `u64`) to/from `[u8; 32]`, the `i64` conversion methods use this method and convert between `i64` and `u64`.

To convert from a `u64` to an `i64`, first the value of the `u64` is checked. If the `u64` value is less than 2^{63} , then it can be directly converted to an `i64`. This is because binary representation of an integer in the range of 0 to $2^{63} - 1$ is equivalent for both types. For an n -bit unsigned integer, the value can be calculated as $2^0 \cdot b_0 + 2^1 \cdot b_1 \dots + 2^{n-1} \cdot b_{n-1} + 2^n \cdot b_n$, where b_i is the value of the i 'th least significant bit. For an n -bit two's complement integer, the equivalent expression would be $2^0 \cdot b_0 + 2^1 \cdot b_1 \dots + 2^{n-1} \cdot b_{n-1} - 2^n \cdot b_n$. As such, as long as the most-significant bit is 0, then the decimal value of an `i64` and a `u64` with the same binary representation is equal. If the `u64`'s most significant bit is 1, then we can convert to two's complement by subtracting 2^{64} . This must be done in a slightly roundabout manner in Rust, by subtracting 2^{63} to convert the `u64` into the

range of an `i64`, and then subtracting 2^{62} twice (as 2^{63} cannot be represented as an `i64`). The reverse conversion from `i64` to `u64` is similar, requiring 2^{64} to be added rather than subtracted.

Static Storage Module Functions

The `Storage` module also provides several static functions. Functions for converting between an `Address` and an array of four `i64`s are provided. To convert from an `Address` to four `i64` values, `address_to_i64` converts the address to a storable `[u8; 32]` array using `Storage::from_address`. This array is split into four separate 32-byte storable values, where the first 8 bytes of each is a quarter of the storable value, and the rest are 0. These can be converted to an `i64` using `Storage::from_i64`. The function `i64_to_address` works similarly in the opposite direction.

The `Storage` module also provides a private function `add_to_key`. This function takes a storage key (as an `H256`) and an unsigned 64-bit integer value, and increments the address by the value. This is done by repeatedly incrementing the least-significant bit of the address, and decrementing the value, until the value hits zero. If the least-significant bit overflows, then it is set to 0 and the next least-significant bit is incremented. This is repeated if the next least-significant bit is incremented. If the second most-significant byte overflows, then an error is thrown as this would violate the separation of memory namespaces mentioned in the *Memory Layout of Smart Contract Storage* segment of this section.

StoresFixed Implementation for `[u8; 32]`

For the `write` method implementation, the storable array is first written into the smart contract storage using `pwasm-ethereum`'s `write` method. After this, the `Storage` struct's cache is checked for the presence of the given key. If the key is present, the value is updated to the new value provided, otherwise it is added to the table. For the `read` implementation, the cache is checked for the presence of the given key. If this hits, then the corresponding value is returned, otherwise it is read from smart contract storage and written into the cache before being returned. The storage address returned from both of these methods, representing the last-used address in writing the given object to storage, is the same as the key passed in - this is because a `[u8; 32]` value can be stored in one storage slot.

StoresFixed Implementation for Primitive Types and Address

As mentioned earlier, the `Storage` struct implements several conversion methods between primitive types (`u32`, `i64`, `bool`) or `Address` values and storable values (`[u8; 32]`). Leveraging this, the `StoresFixed` implementations for these types simply convert the values into a `[u8; 32]` value and call into the `StoresFixed<[u8; 32]>` implementations.

StoresFixed Implementation for `Option<T>`

In order to store values of type `Option<T>` (which take the value `None` or `Some(v: T)`), we must be able to store values of type `T` and a value to indicate whether or not the object has a concrete value. The `write` method for `Option<T>` will pattern match the given value, and write a boolean `true` or `false` depending on whether a concrete value exists. If one exists, it is written to the next slot in storage; if not, nothing is written to the next slot in storage. Either way, the last-used key returned is the passed-in key plus the size of `T` in storage, in order to reserve a second slot in storage regardless of whether a value was written (or else the storage size of `Option<T>` would not be fixed). The `read` method simply checks if a value is stored or not, then reads and returns `Some(value)` or `None`.

StoresFixedVec Implementation

Vectors are stored in smart contract storage sequentially, with the first storage slot containing the number of elements in the vector, and the following slots containing values written using the `StoresFixed<T>` implementation. Vectors are always written in their own memory namespace - i.e. their most-significant byte is unique - to prevent a change in size from overwriting adjacent values in storage.

The `write_vec` implementation gets the length of the given vector and writes that to the first storage slot. After this, the vector is processed to write each element into storage sequentially. The last-used key is kept track of throughout this process, and returned after the whole vector is stored. The `read_vec` implementation is similar, first reading the vector length and then reading the elements sequentially from storage into a new vector, returning the last-used key. The `length` method simply reads the length from the first storage slot of the vector, and `set/get/push` which deal with individual vector elements will find the address of the required index by calculating the starting address plus the size of storing values of type `T` multiplied by the index.

StoresFixed Implementation for Tuples

`StoresFixed` is implemented for tuples with two and three elements, where an implementation of `StoresFixed` exists for all element types. This allows the smart contract to store small generic sets of data without implementing a specific struct, like an observable's arbiter, name, and value. Tuples are stored sequentially in memory, which is trivial using the `write/read` implementations of the tuple element types.

StoresVariable Implementation for Vectors and Observable Names

When the smart contract is constructed, a set of serialized observable names is generated from the given combinator contract. These are represented by a vector of integers per observable name; to store these names, a vector of vectors must be stored. As a vector of integers can grow and shrink, a `StoresFixedVec<Vec<T>>` implementation cannot be used. As such, `StoresVariable` exists to allow writing/reading variable-sized objects to and from storage in their entirety. `StoresVariable` is implemented for `Vec<T>` where `StoresVariable<T>` is implemented, and for `ObsName` - a struct containing the serialized observable name vector. The implementations are similar to the `StoresFixedVec<T>>` implementation, without the element-wise methods.

4.4 Evaluation

In order to evaluate the design of the smart contract (minus combinator implementations), there are a couple of methods that can be employed. Qualitative analysis of the design of the smart contract's ABI can evaluate how well the smart contract fulfils the requirements laid out in section 4.1. Qualitative analysis of the `Storage` module can also be employed to evaluate its design. Automated tests have also been used to ensure that the smart contract operates correctly, as described in section 4.2, and that the `Storage` module operates as expected.

4.4.1 Design of the Smart Contract's ABI

To evaluate the smart contract's ABI, it can be compared to the requirements laid out in section 4.1. The first requirement described the requirement that a financial smart contract should represent an agreement between a *holder* and a *counter-party*, and should allow the two parties to exchange funds. This is implemented in the contract by storing the holder and counter-party's addresses, and the staking and withdrawal of funds from the smart contract allows an indirect exchange of funds between the two parties.

Unfortunately, the smart contract does not facilitate direct payments between the two parties, but this would be impossible as a smart contract cannot initiate a transaction with another user's funds. Another issue with this implementation is that payments do not occur in real time, but instead retroactively through the `withdraw` method. Due to the lack of scheduled callbacks, and the requirement that funds are staked before being sent by a transaction, this would also not be possible. Retroactive payments are the next best option, and are not too dissimilar to the way that some financial contracts would handle transactions, e.g. via an escrow mechanism.

The second requirement was that the holder should be able to *acquire* the contract, which should result in the evaluation of payments between the holder and the counter-party, with respect to acquisition time. This is implemented through the `acquire` method on the smart contract, allowing the holder to acquire it at any point in time, and evaluating payments required between the two parties - represented by a balance for each party. Assuming that the evaluation of these payments is correct (discussed in section 4.4.3), then this requirement is aptly fulfilled.

The third requirement, requiring the evaluation of payments to respect the given financial combinator contract's expected behaviour, is handled by the implementations of the combinators. This is detailed in section 5.

The fourth requirement requires the setting of observable values and `or` combinator branch choices to be possible. The setting of observable values is implemented through the `set_obs_value` method on the smart contract's ABI, which allows pre-defined arbiters to provide the values of observables. Ideally, the observables' values could be obtained automatically by the smart contract, but due to the lack of scheduled callbacks or off-chain interaction this is not possible. The implemented solution is the next best option.

The choosing of `or` combinator sub-contracts is implemented via the `set_or_choice` method on the ABI, allowing the holder to choose which sub-contract to acquire when the given `or` combinator is acquired. This works similarly to how a real financial contract works, where the holder must choose the sub-contract by making a choice manually, and so the requirement is fully met.

The final requirement described earlier is for the smart contract to keep track of the evaluation of payments over time. This is handled by keeping track of the balance of the holder and counter-party, and by updating it retroactively through calls to the `update` method in the smart contract's ABI. This method evaluates the difference in balance due to payments occurring since the last `update` call, thus keeping track of payments over time. Assuming that evaluation of payments is correct, as evaluated in section 4.4.3, then this method of keeping track of payments over time is sufficient to fulfil this requirement.

Based on these requirements - while there are some issues due to the lack of scheduled callbacks, off-chain interactions, and so-on - the final implementation is still sufficient to represent a traditional financial contract as a smart contract with similar behaviour.

Besides these requirements, there is one issue which plagues the smart contract's ABI: serialization. Most methods involve some level of serialization, with serialized arguments or return types. The reason that this is an issue is that it causes the smart contract to be very difficult to interact with from a command-line. Unfortunately, this issue is somewhat unavoidable due to the inability to send strings over a smart contract ABI implemented with `pwasm-abi`; implementing the smart contract in another language, like Solidity, may have avoided this however. The issue is also somewhat assuaged by the existence of the web client, through which most interaction with financial smart contracts should occur. For anyone aiming to write a financial contract, it would likely be preferable to use a web client with lower requirements of technical skill compared to a command-line interface anyway. As such, the problem is not a deal-breaker.

4.4.2 The Storage Module

Another major component of the smart contract implementation is the **Storage** module. While there are no specific requirements of the **Storage** module with regards to the project as a whole, it can still be evaluated in regards to its design and performance.

The use of generic traits to handle the storage ABI results in a simple, easy to use **Storage** struct where most implemented types can be written/read using only one method. This trait design is also reusable, and is not specific to the smart contract storage model; while this is not essential for this project, it is an indicator that the design of the traits handles encapsulation well.

All of the storage trait implementations handle writing and reading in linear time at worst, and constant time at best. The element-wise lookup/update functionality for most vectors is very useful, resulting in most operations which write to storage in the contract taking constant time. In an Ethereum smart contract, extra complexity can result in the user spending more in gas fees to call a function - the more operations a function call carries out, the higher the gas cost in Ether. While this project may not be ideal for use on public blockchains, and may be more useful in the context of private blockchains within/between financial organisations where gas is less of an issue, the lower usage of resources is a benefit no matter what. While variable-sized values must be read/written in their entirety, taking linear time, this is not a huge issue - the only variable-sized value stored in the smart contract is the vector of observable names. This is only written to at the beginning of the contract, and always read and returned in its entirety when `get_obs_entries` is called - the latter usage is part of a pure function, however, and thus costs no gas anyway. As such, the impact is minimal.

One potential area for improvement in the **Storage** module is the amount of storage used. For example, storing an `Option<i64>` requires storing a `bool` in one storage slot, followed by an `i64` in the following storage slot. In total, this takes 65 bits. A smart contract storage slot stores up to 256 bits of information, so this information could be stored in 1 storage slot. Because of the generic trait system, it is difficult to optimise the amount of space objects with sub-values take. Furthermore, it would require much more specific implementation based on the type of the value being stored, which would quickly become complex and unwieldy, and require much more effort to implement. As such, the current implementation is preferred, but optimisations - like reducing storage sparsity by serializing combinators directly to byte arrays instead of integers - could definitely be implemented.

Another interesting aspect of the **Storage** struct is that it implements a key/value cache. The aim of this is to reduce the performance hit of reads when the same address is read from multiple times in one financial contract ABI call. One issue with this is that in general, no ABI call will write to/read from the same address more than once; acquiring and updating the contract only updates each combinator once, and other function calls are not iterative or recursive besides the constructor. This could potentially mean that the performance hit of maintaining the cache does not have enough of an upside to make it worthwhile. This is difficult to evaluate concretely, but it is possible that the implementation of a cache is not improving efficiency in this context.

4.4.3 Automated Testing

In order to ensure that the behaviour of the smart contract is correct, the most obvious tool to use would be automated testing. A suite of unit tests was written for the main smart contract module, as well as a unit test suite for the **Storage** module, and a set of integration tests for the smart contract, **Storage**, and combinator modules.

One important note is that for any of the smart contract tests written in Rust, the top-level smart contract struct was not deleted between function calls (which would occur in real-world operation on a blockchain). This means that all reads from storage would hit in the **Storage** struct's cache, and so reading from and writing to actual smart contract storage are not tested in Rust. The

conversion between types is still tested, however, and the writing to/reading from storage relies on the correctness of the `pwasm-ethereum` storage methods - as such, this is an acceptable compromise. There are also JavaScript integration tests for the smart contract, which *would* have the smart contract struct deleted between function calls - thus requiring the use of smart contract storage. Besides this, there should be no meaningful differences between the Rust test environment and real-world operation.

Smart Contract Unit Tests

The general testing philosophy for the smart contract ABI functions is fairly simple; for every function in the ABI, the expected functionality should be fully covered by tests, and cases where errors are thrown should all be tested. The `pwasm-test` module enables unit testing of the smart contract with mocking of calls to `pwasm-ethereum` (e.g. to find the sender of the contract call transaction). This was used with Rust's normal testing framework for all unit tests. There are 37 unit tests made up of 650 LoC (Lines of Code) for the top-level smart contract module.

To ensure that the holder, counter-party, and contract definition are correctly stored after the constructor, tests on the top-level getter functions for these values have been implemented.

The `update` method is tested to ensure that it does nothing before acquisition, calls into the combinators correctly after acquisition, and sets the last-updated time. The calculation of the balance of the two parties is also checked in one test each. Assuming that the combinators operate correctly, this covers its functionality. The `acquire` method is tested alongside the `update` and `get_acquisition_times` methods, as it does not change state in an externally-visible way on its own.

The setters and getters for observables and `or` combinator choices, and `anytime` combinators' acquisition method are all tested by calling the setters/acquire method, and checking the resulting observable values, `or` choices, and `anytime` acquisition times respectively.

The helper function for the `withdraw` method is checked to ensure that the correct withdrawal amount is calculated under all special cases, i.e. with sufficient funds/balance and with/without gas fees, with insufficient funds and with/without gas fees, with insufficient balance and with/without gas fees, and with balance/funds below the gas price when gas fees are not used.

Besides these functionality tests, errors in the main smart contract module are tested under conditions where the errors should be thrown, and by expecting the error. If no error is thrown, or the error thrown does not match the expected error string, the tests fail. All potential errors in the smart contract are tested this way.

Storage Unit Tests

In order to ensure that the smart contract operates correctly, it must be the case that the writing and reading of data to and from storage is correct, i.e. that after writing a value to a storage address, reading from the same address returns an equivalent value. In order to test this, each storage trait implementation is tested, as well as the public functions to convert between addresses and signed 64-bit integers. There are 15 unit tests made up of 150 LoC for the `Storage` module.

For each `StoresFixed<T>`, `StoresFixedVec<T>`, or `StoresVariable<T>` implementation, a test function exists which writes several values of the type `T` to storage, and then reads them back. The original values and the values read from storage must be equal, or else the test fails. For `StoresFixed<T>`, the `size` functions are also checked.

Integration Tests

There are several integration tests written in Rust, designed to ensure that the behaviour of the contract is correct throughout multiple method calls. The majority of these tests involve supplying a contract definition, supplying any required inputs, and acquiring/updating the contract - the final balances are then checked to ensure that contract evaluation acts correctly as a whole. In total, there are 25 Rust integration tests made up of 400 LoC, and 25 JavaScript integration tests made up of 350 LoC.

There is at least one Rust integration test for each combinator, involving contracts written using the combinator in question. Evaluation of the behaviour of the combinators is left for section 5.3, these tests instead focus on checking of the entire flow of the smart contract's execution. For an example, the test for the smart contract with the combinator contract `one` supplies the contract definition, acquires the contract, and checks the balance. This tests that the flow through the smart contract over time is correct.

Several of these integration tests require the use of extra smart contract ABI methods. The `set_or_choice` method is tested for left and right or choices, `truncate` is tested with the acquisition time before and after the horizon, `set_obs_value` is tested with a contract with an observable, and `acquire_anytime_sub_contract` is tested with a contract with an `anytime` combinator. `get_concluded` is also tested, both by expiration and by updating.

Besides the Rust integration tests, there is also a large suite of integration tests written in JavaScript. These tests deploy the smart contract to a private Ethereum blockchain node (running in the Parity client), and make calls to the contract using the Web3.js library. While these tests are stored with the tests for the web client (for ease of running tests/building), they only test the smart contract functionality.

These JavaScript tests are written similarly to the Rust integration tests; for each type of combinator, there are tests of its expected behaviour. There are also tests which ensure that payment in Ethereum operates correctly, ensuring that the smart contract obtains staked Ether, and that the relevant parties obtain Ether when withdrawing it.

4.4.4 Conclusion

Overall, the smart contract's ABI fulfils the requirements laid out in section 4.1 to the best level that can be achieved given the limitations of Ethereum smart contracts, or at least close to it. The `Storage` module is fairly efficient and well designed, but the cache implementation may not be as beneficial as one would hope. Thorough automated testing helps to support the correctness of these modules, all of which pass.

4.5 Remarks

In this chapter, the design and implementation of the smart contract which can represent any SmartFin contract has been described in detail. The final result is a smart contract which implements all of the required functionality of a SmartFin contract about as accurately as possible - including an ABI for interacting with the smart contract, and persistent storage of the smart contract state - with few issues.

While this section covers most of the smart contract implementation, the implementation of the SmartFin combinators' behaviour has not yet been described; in the following chapter, the smart contract representation of these combinators is laid out in detail.

Chapter 5

Designing and Implementing SmartFin Combinators

The top-level representation of SmartFin financial contracts by a generic smart contract has been described already in chapter 4, but the actual semantics of each combinator must also be represented in this smart contract. This chapter describes how the semantics of SmartFin’s combinators are implemented. The modules covered in this chapter are displayed in figure 5.1 as a dependency graph.

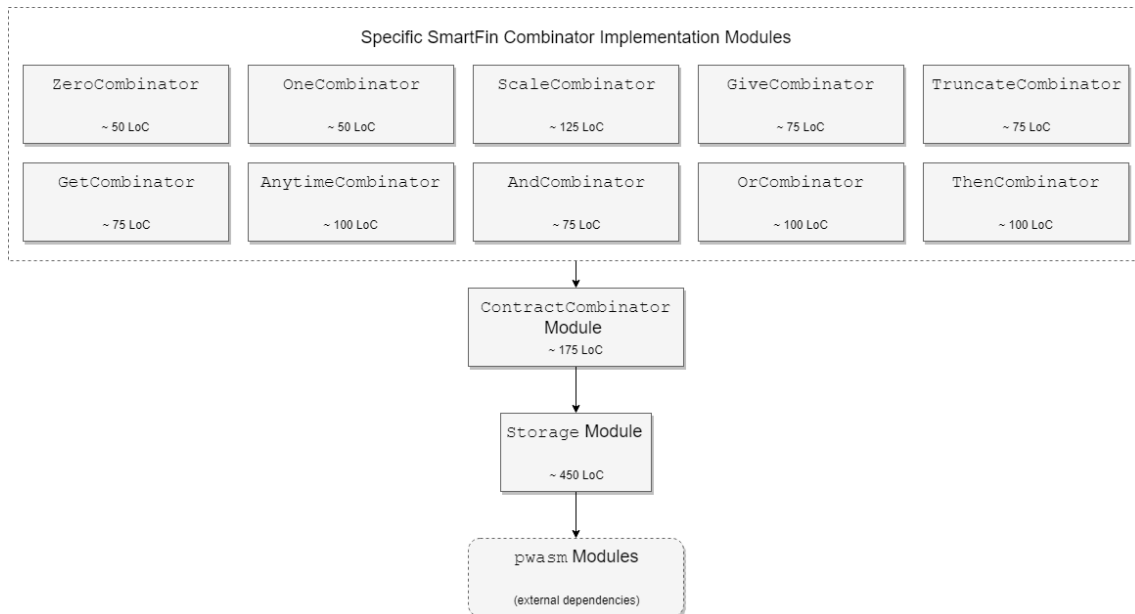


Figure 5.1: A dependency graph of the modules implemented to represent the SmartFin combinator semantics, with their approximate *Lines of Code* (not including tests). **pwasm** modules are external dependencies, described in section 4.3.1. The **Storage** module is described in section 4.3.4.

5.1 Designing a Programmatic Representation of SmartFin Combinators

5.1.1 General Representation of Combinators

As described in chapter 3, SmartFin is functional and compositional by definition. In any proposed smart contract implementation of SmartFin, representing the combinators in a functional manner is not simple; the smart contract must keep track of state over time, and individual combinators will have different state to keep track of. This suggests that a purely functional representation of the combinator behaviour may not be ideal, as keeping track of state could be complicated and unwieldy.

Furthermore, each combinator's functionality cannot be described simply as a single function - for example, how can we represent the `get` combinator, which acquires its sub-combinator at its horizon, without scheduled callbacks (which aren't possible within Ethereum smart contracts)? It makes more sense to represent it with an acquisition function and an `update` function which is called at a later date.

Due to the requirements of keeping track of state, and declaring multiple functions over this state, it would make sense to represent the combinators as classes rather than functions. As such, each combinator has a struct and a trait implementation. Due to the modular nature of the combinators in SmartFin, the information each combinator requires about their sub-combinators is minimal - there are no cases where certain combinations of combinators have non-standard behaviour. Because of this, the combinators implement a standard trait (`ContractCombinator`) for their functionality.

While the combinators cannot be easily represented in a smart contract in a functional manner, this does *not* mean that they cannot be represented in a *compositional* manner. In SmartFin, combinators may have a number of sub-combinators; this can be represented as a tree of combinator objects, where each object may have zero, one, or two sub-combinator objects. This would be simpler than using some other data structure (e.g. a vector) to store the combinators, as it removes any extra requirements of external knowledge; each combinator object will only know about its sub-combinator objects, and nothing else. It also makes logic for traversal by recursion simple, intuitive, and efficient.

In summary, the final design for the representation of SmartFin's combinators in the smart contract is a tree of objects. Each object is a struct which implements the `ContractCombinator` trait, allowing combinators to call methods on sub-combinators without knowing which combinators they represent. Each combinator will store its own sub-combinators in its struct, and method calls will be propagated by recursive calls to the sub-combinators. Each combinator's behaviour will be implemented entirely within its struct's method implementations.

To illustrate this idea, take the SmartFin version of the financial contract representing a European option from section 2.3.3, `get truncate <01/01/2020 00:00:00> or scale 500 one zero`; the representation of these combinators as a tree of struct instances is depicted in figure 5.2.

5.1.2 ContractCombinator Methods

The `ContractCombinator` trait is a trait that all combinator structs implement, with methods that are required regardless of the combinator type. This enables combinators to call methods on their sub-combinators regardless of the sub-combinators' types, and the same for the top-level smart contract. These methods are as follows:

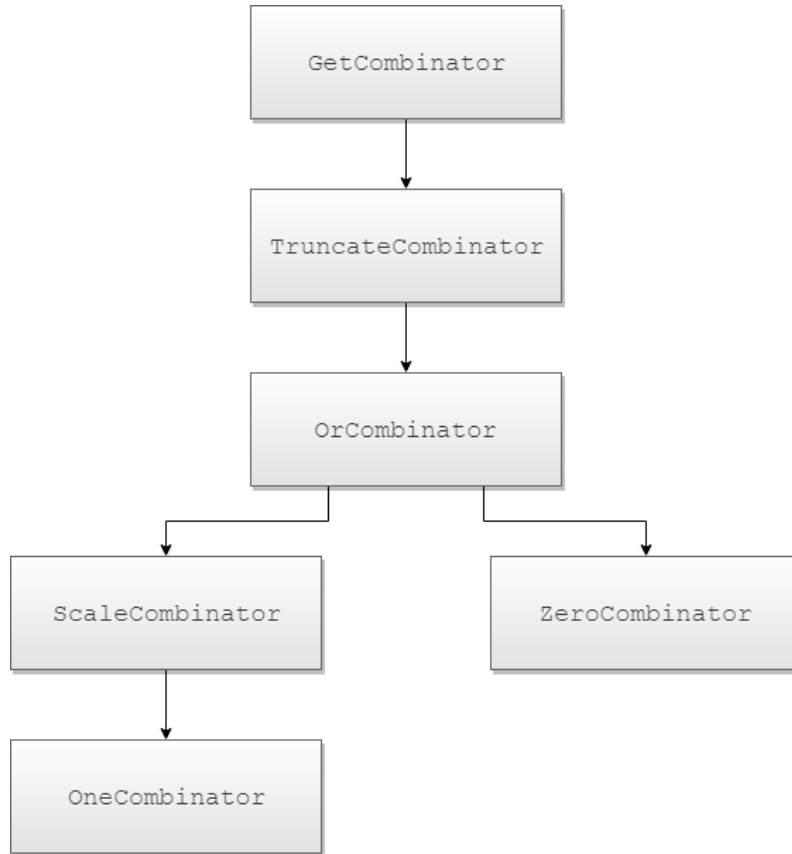


Figure 5.2: The representation of the European option SmartFin contract, `truncate <01/01/2020 00:00:00> or scale 500 one zero`, as a tree of combinator struct instances.

- `get_horizon`: returns the combinator's horizon as an `Option<u32>`, with the value `Some(<UNIX Epoch time>)` if a horizon exists, or `None` otherwise.
- `past_horizon`: returns true if the given time (in UNIX Epoch time) is beyond the combinator's horizon.
- `acquire`: handles the acquisition of the combinator (and sub-combinators if applicable).
- `update`: handles updating the combinator to the current time, mainly to evaluate payments that occur since the last call to `update`. The method returns the amount of funds in Ether that the counter-party should have paid the holder in this period, as a signed 64-bit integer (where a negative result signals payments from the holder to the counter-party).
- `get_combinator_number`: returns the combinator's enum value.
- `get_combinator_details`: returns a `CombinatorDetails` struct, containing the combinator's acquisition time as an `Option<u32>`, and whether or not it has been fully-updated as a `bool` (i.e. further calls to `update` will have no effect). These details can be used by parent combinators, or by the top-level smart contract - i.e. to check if the contract has concluded or been acquired.
- `serialize`: returns the combinator (and sub-combinators) in a serialized format (`Vec<i64>`), which is required to store the combinator tree in smart contract storage.
- `serialize_details`: returns the combinator's `CombinatorDetails` struct in a serialized format (`Vec<i64>`), which is called when serializing the combinator. This is a trait method to provide a general implementation of this method, given that the `CombinatorDetails` objects have the same structure regardless of the combinator type, and all combinators implement the `get_combinator_details` method.

Besides these trait methods, each struct implementing `ContractCombinator` also implements a `deserialize` constructor. These are not present in the `ContractCombinator` trait because they are static - as there is no combinator object until after deserialization - and static methods cannot be defined for traits in Rust. This method takes in a vector representing the combinator tree, with all of its state, in a serialized format. It also takes in an index, which is the location of the combinator being deserialized in the vector. The method returns the deserialized `ContractCombinator` instance and the last index used during deserialization. The index is needed for combinators like `and`, where the starting index of the second sub-combinator is not known until the first sub-combinator is deserialized.

5.2 Implementation

Every combinator has their own struct and implementation of the `ContractCombinator` trait, and the `ContractCombinator` module has some extra functionality for utility.

5.2.1 The ContractCombinator Module

The `ContractCombinator` module defines default implementations of some methods for the `ContractCombinator` trait, as well as some utility methods.

Default Method Implementations for the ContractCombinator Trait

Default implementations are not provided for the `acquire`, `update`, and `get_combinator_number`, as these implementations obviously depend entirely on which combinator is being implemented. A default `get_combinator_details` method is also not implemented, as there is no way to declare that structs will have a specific member variable when implementing a trait (as opposed to something like a Superclass in other languages). As such, there is no way to obtain the `ContractDetails` struct from the member variable as far as the `ContractCombinator` trait is aware, and so each combinator struct must implement their own method to retrieve this variable. This is a specific limitation with Rust. In all combinators, the methods `get_combinator_number` and `get_combinator_details` have a trivial implementation, and so they are not mentioned further.

The default `get_horizon` implementation returns `None`, as any combinators without sub-combinators will have no horizon, and so it is the basest implementation of the method where no extra information about the combinator type is required. The default `past_horizon` implementation gets the horizon with `get_horizon`; if the horizon is `None`, then the horizon can never be passed, otherwise it is compared to the given time. This is never overridden by any `ContractCombinator` implementations, as it will never change as long as `get_horizon` is accurate.

The default `serialize_details` implementation pushes the combinator enum, the acquisition time (represented as the value, or -1 if no value exists), and whether or not the combinator is fully-updated (1 represents `true`, 0 represents `false`). These are obtained by the `get_combinator_number` and `get_combinator_details` functions. This is standard for all combinators, and is used in the `serialize` method.

The `serialize` method is required to store the combinator and its state in the smart contract storage between function calls, and serializes this information into a vector of signed 64-bit integers (or `i64s`). The default implementation of this method simply returns the return value of `serialize_details`; this will represent any combinators with no extra state, and the other combinators will have to serialize their own state.

Other Utilities in the ContractCombinator Module

Besides the trait implementation, the `ContractCombinator` module also provides several utilities. One such utility is a `Combinator` enum representing the contract combinators defined in chapter 3, and implementations for converting between this enum and an integer (for serialization purposes).

The `CombinatorDetails` struct definition is also provided here, containing the acquisition time of a combinator, and whether or not a combinator is fully-updated. All combinators are able to return their `CombinatorDetails` struct; `fully_updated` is used to prevent calling into combinators for no reason, and `acquisition_time` is used to ensure that combinators have been acquired (especially `anytime` combinators) before being called into. Two methods for initialising a `CombinatorDetails` struct are also provided, one with default attributes (no acquisition time and not fully-updated), and one from the serialized format of the `CombinatorContract` struct.

The `ContractCombinator` module also provides two public functions for comparing times; `earliest_time` will take in two times and return the value of the earliest, and `latest_time` will take in two times and return the value of the latest.

The module also provides a function for deserializing combinators - given a serialized combinator vector and a starting index, the function will check the serialized combinator's enum and initialise a combinator object of the correct type using the correct `deserialize` constructor.

5.2.2 Simple ContractCombinator Implementations

Several of the `ContractCombinator` implementations have simple implementations throughout all of the combinator modules. The `get_combinator_number` method returns the value in the `Combinator` enum representing the implementing combinator's type. The `get_combinator_details` method just returns the `CombinatorDetails` object from struct.

`get_horizon` typically returns `None`, or the latest horizon of any sub-combinators (except in the case of `truncate`). `serialize` will write the result of `serialize_details` to a vector, followed by any struct variables, and then any sub-combinators sequentially. While it isn't on the trait, all combinators have a `deserialize` function which reads from a serialized vector as described and reconstructs the combinator struct.

Besides these methods, all combinators implement their own `acquire` and `update` methods as described below. All of these `acquire` methods will throw an error if the combinator is previously acquired or expired, and the `update` methods will return 0 balance change if the combinator is not acquired in the past or is fully updated.

5.2.3 The ZeroCombinator and OneCombinator Modules

The `acquire` method for both the `ZeroCombinator` and `OneCombinator`'s implementations of `ContractCombinator` both set the combinator's acquisition time to the current time. For the `update` method, both combinators mark themselves as fully-updated, `ZeroCombinator` returns 0 as the evaluated difference in balances, and `OneCombinator` returns 1.

5.2.4 The AndCombinator Module

The `acquire` method for the `AndCombinator` checks if the sub-combinators' horizons have passed yet. For each sub-combinator, if the horizon has not passed then its `acquire` method is called with the same parameters as the `AndCombinator`'s. The `AndCombinator`'s acquisition time is then set

to the current time.

The `update` method calls `update` on the two sub-combinators in order. If both of the two sub-combinators are fully-updated after their `update` calls, then the `AndCombinator` is set as fully-updated. The sum of the values returned by the sub-combinators is returned.

5.2.5 The OrCombinator Module

The `OrCombinator` struct is one of the more complicated combinator implementations, requiring the holder to select which of its two sub-combinators should be acquired. The struct contains the combinator's `or` index. This is the index of the `or` combinator in the combinator contract, with regard to all `or` combinators, ordered sequentially by left-to-right occurrence and starting from 0. This is used to index into a vector of all `or` combinator choices in the contract, stored in the `Storage` struct described in section 4.3.4.

A method `get_or_choice` is provided on the struct, which determines which sub-combinator should be acquired when the `or` combinator is acquired. The result is represented by an `Option<bool>` value, where `Some(true)` represents the first sub-combinator, `Some(false)` the second, and `None` that neither sub-combinator should be acquired. If either sub-combinator is expired, a value signalling that the other should be acquired is returned. If neither is expired, the `or` choice is looked up and returned from storage.

The `acquire` uses `get_or_choice` to check which sub-combinator to acquire. If `Some(val)` is returned then the relevant sub-combinator's `acquire` method is called and the `OrCombinator`'s acquisition time is set to the current time, otherwise nothing occurs.

The `update` method obtains the `or` choice by calling `get_or_choice` with the `OrCombinator` instance's *acquisition* time, *not the current time*. If the sub-combinator to have acquired is ambiguous, the method returns 0. If a concrete `or` choice is returned, the method checks if the relevant sub-combinator has been acquired; if not, then its `acquire` method is called with the `OrCombinator`'s *acquisition* time. After this check, the `update` method on the relevant sub-combinator is called and returned, and the `OrCombinator` is set to fully-updated if the relevant sub-combinator is also fully-updated.

5.2.6 The GiveCombinator Module

The `acquire` method simply acquires the sub-combinator and sets the acquisition time. `update` simply updates the sub-combinator and returns the negated result. The combinator's fully-updated flag is set to the value of the sub-combinator's fully-updated flag.

5.2.7 The ScaleCombinator Module

The `ScaleCombinator` is one of the more complicated combinator implementations. This is because a `scale` combinator scales the sub-contract's value by an observable. This observable can be a fixed constant, or it can be a value that varies over time. Both cases must be handled by this combinator. The `ScaleCombinator` struct has a scale value, an `Option<i64>` that represents the `scale` combinator's observable when it is a constant value. The struct also has an observable index, an `Option<usize>` representing the `scale` combinator's observable's index in a set of time-varying observable values. One of the scale value or observable index must have a concrete value, i.e. `Some(v)`, and the other must have no concrete value, i.e. `None`.

The method `get_scale_value` is implemented on the `ScaleCombinator` struct. This method checks if a concrete scale value exists; if so, it is returned, otherwise the observable index is looked

up through the `Storage` struct and returned (as an `Option<i64>` which may not yet have a concrete value). If neither a concrete scale value or observable index exists, an error is thrown.

The `acquire` method is simple, simply acquiring the sub-combinator and setting the acquisition time. The `update` method gets the scale value using `get_scale_value`. If no concrete scale value exists yet then the value 0 is returned, otherwise the sub-combinator is updated, and fully-updated is set if the sub-combinator is fully-updated. The sub-combinator's update value is then multiplied by the scale value and returned.

5.2.8 The TruncateCombinator Module

The `CombinatorContract` implementation for the `get_horizon` implementation returns the earliest of the `TruncateCombinator`'s truncated horizon and the sub-combinator's horizon - as described in section 3.3. The `acquire` method simply checks the horizon as usual, acquires the sub-combinator, and sets the acquisition time. `update` performs the usual call-through to the sub-combinator, setting fully-updated to the sub-combinator's fully-updated value and returning the result from `update` on the sub-combinator.

5.2.9 The ThenCombinator Module

The `acquire` method checks if the current time is past the first sub-combinator's horizon. If not, the first sub-combinator is acquired, otherwise the second is acquired, and then the acquisition time is set. The `update` method checks if the `ThenCombinator`'s acquisition time is past the first sub-combinator's horizon (as before). If not, the first sub-combinator is updated, otherwise the second is updated. Fully-updated is then set based on the sub-combinator, and the sub-combinator's update value is returned.

5.2.10 The GetCombinator Module

The `GetCombinator` struct is another relatively simple `ContractCombinator` struct, although it may seem unintuitive at first. The `acquire` method carries out the main behaviour of the `get` combinator. The sub-combinator's horizon is checked, and if a concrete horizon is found then the sub-combinator is acquired *with the horizon as the acquisition time*. This will cause the sub-combinator to only become updated once the horizon is reached or passed, as is required by the definition of the `get` combinator in section 3.3. The `update` method is simple, calling through to the sub-combinator and setting fully-updated and returning the sub-combinator's `update` result as usual.

5.2.11 The AnytimeCombinator Module

The `AnytimeCombinator` is one of the more complex combinators, allowing the acquisition of the sub-combinator at any point until its horizon (as described in section 3.3). The `AnytimeCombinator` struct has an `anytime` index. This is the index of this `anytime` combinator's sub-contract acquisition time, in a vector of `anytime` acquisition times stored in smart contract storage.

The vector contains `(bool, Option<u32>)` tuples - the boolean value represents whether or not the corresponding `anytime` combinator has been acquired, and the `Option<u32>` represents the prospective acquisition time of the sub-combinator. If this time is in the future, it can be changed by calling the `acquire_anytime_sub_contract` method on the smart contract's ABI, setting it to the current time as long as the parent `anytime` combinator has been acquired. If the prospective acquisition time is in the past, then it cannot be changed.

The `acquire` method sets the `AnytimeCombinator`'s acquisition time and sets the prospective acquisition time of the sub-combinator in the stored `anytime` acquisition times vector to the sub-combinator's horizon. This means that the sub-combinator can be acquired manually up until its horizon, after which its acquisition time will be locked in as the horizon. We cannot call `acquire` in this method on the sub-combinator, as a combinator can only be acquired once and the sub-contract's acquisition time may change in the future.

The `update` method checks if the sub-contract has been acquired yet. If not, then the `anytime` acquisition times vector is checked for its prospective acquisition time. If the sub-combinator's acquisition time has passed then `acquire` is called on it. After this, `update` calls through to the sub-combinator as usual.

5.3 Evaluation

In order to evaluate the representation of the SmartFin combinators in the implemented smart contract, the design of the combinators' representation in the smart contract can be qualitatively evaluated, and automated testing can provide some quantitative evaluation of their correctness.

5.3.1 Combinator Representation

Representing the combinators as a tree of `ContractCombinator` structs results in an intuitive composition of combinators that behave as one would expect based on the SmartFin contract definition. Combinators apply behaviours over their sub-combinators, and then call through to the sub-combinators; a financial contract written in SmartFin would also apply behaviour and propagate acquisition to sub-combinators, and so this representation is a natural fit.

One potential implementation approach that may also seem like a natural fit is the implementation of each combinator as its own smart contract. This would allow a SmartFin contract written to be represented as a set of linked smart contracts. Calling methods on each smart contract would call through to the sub-combinator's smart contract. This may seem like a closer match to SmartFin's semantics, as any financial smart contracts would be able to become sub-contracts for other financial smart contracts. There are a few issues with this design, however. One issue is that either state would have to be replicated across every smart contract (e.g. the holder and counterparty address), or it would have to be passed across smart contract boundaries with every function call, which would become difficult to implement in a trustworthy manner (e.g. verifying that the holder is correct). Another issue is that traditional financial contracts are a "package deal" - either you acquire the whole contract, or none of it. Representing each combinator as a smart contract could make it difficult to prevent any combinator in the financial contract being acquired without its parents. Circumventing this would require some path representing the parent-child relationship between combinators to be marked through the smart contracts, somewhat defeating the purpose of this representation. Gas costs of deploying large financial smart contracts would also be much higher with multiple smart contracts. For these reasons, using a single smart contract to represent a SmartFin contract is preferable.

5.3.2 Design of the ContractCombinator Trait

On the whole, the design of the `ContractCombinator` trait facilitated the operation of the combinators fairly well, and the chosen methods make contract acquisition/updating simple in general. The ability to get a combinator's horizon, or check if a time is past its horizon, is effectively used by every combinator with sub-combinators; as such, the `get_horizon` and `past_horizon` methods are a useful part of the trait. The ability to serialize combinators relatively painlessly with `serialize_details` and the guaranteed `serialize` method allowed the serialization of the combinators to operate intuitively and efficiently through recursion. Unfortunately, the trait did not provide such methods for deserialization as there can be no static methods defined on a trait. As such, the

`deserialize` constructor was implemented on each combinator struct - this is a limitation of Rust.

Another slight issue with the `ContractCombinator` trait is the `get_combinator_details` and `get_combinator_number` methods. Every combinator has an implementation of these methods, despite the fact that the implementations are extremely similar. Both of these methods could be implemented on the `ContractCombinator` trait easily if there were a way to require the `ContractCombinator`-implementing structs to have certain member variables (like a Superclass); unfortunately, Rust does not provide such a mechanism, and so every struct must implement a method for this. Both methods are required to obtain information needed for serialization and for acquisition/updating the contract, so the current solution is currently the best option available in Rust.

Some possible changes that could be made to `ContractCombinator` involve defining more getter functions, including a function that would return a vector of sub-combinators (which would have 0, 1, or 2 elements). This would allow default implementations of many of the simpler methods in the `ContractCombinator` trait which typically call through to the sub-combinators, e.g. `get_horizon` and `serialize`. Overall, this could potentially reduce the amount of repeated code required between the combinator implementations. Unfortunately, as mentioned already, there is no way to require the definition of a member variable on a struct implementing a trait in Rust. This means that instead of writing repeated implementations for `get_horizon` and other simple methods, there would be more repetition of these getter functions. Unless some method of requiring struct members is implemented in Rust, the benefits to such a change would thus be minimal, hence why they went unimplemented.

While the recursive behaviour of the `acquire` method is a natural fit for the combinators, one possible issue with this design is that the expected behaviour of acquiring the SmartFin combinators has been split across two different methods - `acquire` and `update`. For example, typically acquiring the `one` combinator would result in a payment occurring from the counter-party to the holder; with the current implementation, acquiring the `one` combinator records no payments until `update` is called. This is more of a ramification of the design of the smart contract ABI than a direct design decision; the `update` method is implemented to keep track of payments over time, whereas the `acquire` method simply exists to denote acquisition times. Furthermore, the smart contract ABI updates the balance of the contract in its `acquire` method, nullifying this issue from any external point-of-view. Besides this, the need for an `update` method is obvious given the lack of scheduled callbacks/persistent operation of the smart contract, as explained in section 4.2. These reasons justify the compromise of separate `acquire` and `update` methods.

5.3.3 Design of the Combinators' Implementations

Most of the combinators implement fairly trivial or straightforward behaviour, and as such there is little to discuss regarding their design.

The decided-upon implementation of the setting of observables' values is somewhat interesting. Due to the nature of Ethereum smart contracts, it is impossible to read observables' values from off-chain data. As such, the only way to obtain these values is to have them passed into the contract by an external user, or obtained from another smart contract (eventually requiring external user input).

In the definition of the original DSL by Peyton Jones et al.[27], an observable is defined as something with a value that both the holder and counter-party would agree upon. This definition was considered for the setting of observables, leading to a design where both parties would need to provide a matching value for an observable to set its value. After a discussion with Dr. Panos Parpas of Imperial College London, who wrote financial contracts professionally in the past, this design was passed over in favour of letting a pre-defined arbiter set the value of an observable. This more closely matches the way that traditional financial contracts handle observables, where typically a pre-defined information source for the value of any observable will be included in the

financial contract. This also prevents parties from lying about observable values to avoid unfavourable transactions - although, as described in 4.2, it is impossible to bind the parties to a smart contract's terms without external tools.

5.3.4 Testing

In order to ensure that the combinator implementations behave correctly, extensive unit testing has been implemented to test the functionality of all structs implementing `ContractCombinator`. The tests follow the same structure for all combinators, but some combinators require extra tests. In total, there are 151 unit tests for the combinator modules, made up of 2750 LoC. Besides the unit tests mentioned here, all combinators are tested in the Rust and JavaScript integration tests mentioned in section 4.4.3.

ContractCombinator Module Tests

The `ContractCombinator` module provided several utilities, for which unit tests have been written. The default implementations of the `ContractCombinator` methods were tested by implementing a `DummyCombinator` struct, which only implements methods with no default implementation.

The two time comparison methods, `earliest_time` and `latest_time`, are both tested with concrete values and `None` values for time. The methods for converting between the `Combinator` enum and integer values is tested for all of the enum values. The `serialize_details` method is tested on the `DummyCombinator`, as well as the `serialize` implementation. The `deserialize_combinator` method is tested by defining a combinator tree with all combinator types present and random state, and serializing it. This serialized tree is then deserialized and serialized again; if the second serialized vector is equal to the first, then the deserialization method works correctly - assuming that the serialization methods work correctly. The `deserialize_combinator` function is also tested with an empty serialized combinator vector, to ensure that the correct error is thrown.

General Combinator Testing Methodology

All of the combinators are unit-tested, mostly following the same testing methodology. For each combinator, the `get_combinator_number` and `get_horizon` methods are tested. For combinators with sub-combinators, all variations of sub-combinator horizons are tested, e.g. first sub-combinator's horizon is later, second is later, both are undefined, etc.

The `acquire` method is also checked to ensure that it sets the `CombinatorDetails` struct's acquisition time member correctly, and `update` is checked to ensure that the struct's fully-updated flag is set when required. The value returned by calling `acquire` and then `update` is checked in all permutations (e.g. expired, non-expired, one sub-combinator expired, etc.). The `update` method is tested before `acquire` is called, or before the acquisition time is reached, to ensure that it does nothing.

The `serialize` method is tested by ensuring that all relevant information is encoded in the serialized combinator vector, where the combinator implements extra serialization beyond the default implementation. The `deserialize` constructor is tested by serializing, deserializing, and re-serializing the combinator (as with the `ContractCombinator` tests). Any errors which should be thrown are also tested. This covers the basic operation of all combinators, and only a few combinators require extra testing beyond this.

Specific Combinator Tests

The `OrCombinator` implementation is tested with the left and right sub-combinators chosen or expired for every testing scenario, as well as several tests where no choice is made yet. The imple-

mentation is also tested to ensure that behaviour remains consistent, i.e. you cannot choose one sub-combinator and then see the results of the other sub-combinator after updating.

The **ScaleCombinator** implementation is tested with constant scale values, defined observables, and undefined observables. It is also tested without either is set, to ensure that an error is thrown.

The **AnytimeCombinator** implementation is tested to ensure that acquisition of the sub-combinator occurs on calls to **update**, and that this is set to the horizon correctly. The **update** method is also tested by simulating acquisition of the **anytime** sub-contract, to ensure that it is acquired at the correct time, and that the value returned is correct. Tests with various combinations of acquisition and updating are implemented to ensure correctness and consistency of the acquisition/payment evaluation behaviour. It is also checked that the **AnytimeCombinator** throws the correct error when the **anytime** sub-contract acquisition time is before the parent **AnytimeCombinator**'s acquisition time.

5.3.5 Conclusion

Overall, the combinators' behaviour seems to follow the specification described in section 3.3 based on the thorough unit and integration testing. The design is relatively representative of the SmartFin combinators, although some compromises had to be made due to the nature of smart contracts (like separate **acquire** and **update** methods). The design of the **ContractCombinator** trait covers all of the required functionality, and allows combinator implementations to be relatively simple on the whole.

5.4 Remarks

This chapter has described how SmartFin's combinators are represented in the implemented smart contract. The final result of this and the previous chapters is a smart contract implementation which can represent any given SmartFin contract, with minimal concessions made due to the nature of smart contracts.

While this is useful, we are still missing easy ways of creating, deploying, evaluating, and interacting with financial smart contracts; the next chapter will describe the web client created for this purpose.

Chapter 6

The Financial Smart Contract Web Client

One desired benefit of this project's contributions is the simplification of the creation of financial contracts. While the implementation of the financial smart contract program does simplify this, it is difficult to interact with due to the heavy serialization required in the smart contract's ABI. Rather than requiring users to learn and implement serialization/deserialization of all function parameters and return values, a web client has been implemented to facilitate this interaction and provide some additional functionality.

The web client facilitates the composition, deployment, and interaction/monitoring of financial smart contracts, as well as some basic evaluation functionality - a feature provided to reduce the risk of writing financial contracts and smart contracts. This chapter will describe the design and implementation of this web client. The implemented modules discussed in this chapter are depicted as a dependency graph in figure 6.1.

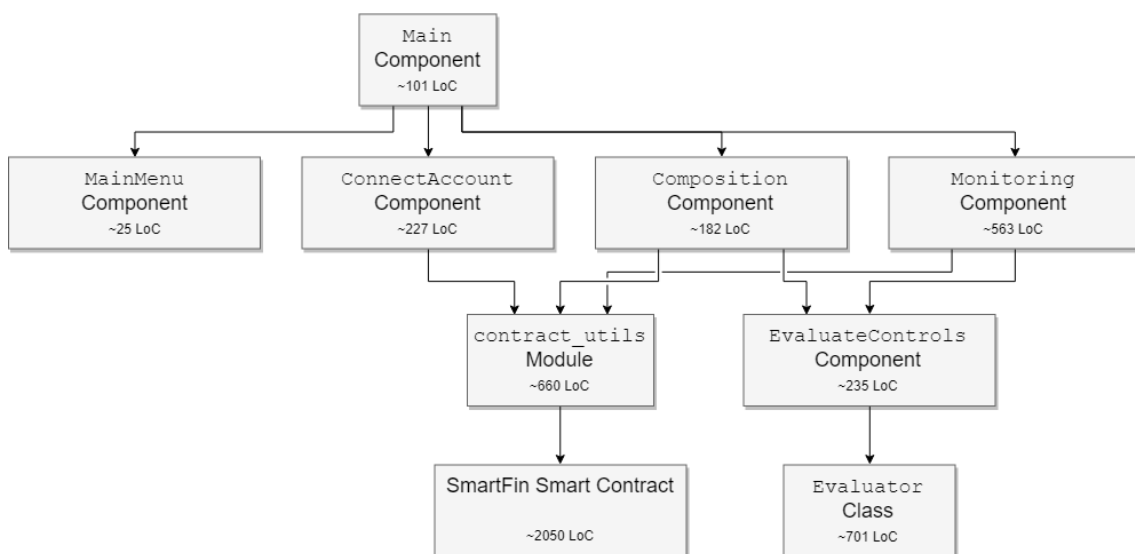


Figure 6.1: A dependency graph of the modules implemented for the web client, with their approximate *Lines of Code* (not including tests). Small UI and data classes are not depicted. The SmartFin smart contract is described in chapters 4 and 5.

6.1 The Design of the Web Client

6.1.1 Architecture

From the beginning, the web client was designed around the use of *React*[24] - a JavaScript UI framework. Using some framework was effectively a necessity in order to design the web client with some semblance of structure, and pure HTML and JavaScript would be significantly harder to coordinate. React was partly chosen due to its relative simplicity, but also due to previous development with it - these factors enabled rapid development of the web client. This is important as the web client UI is intended to be a more basic vessel for specific financial smart contract functionality, not an interesting technical product of its own right; as such, focusing too intently on the web client UI would not be the best use of time.

The design of the web client follows React's typical structure of designing UI views as components, which can be as large as an entire application, or as granular as a single label. Each component has certain *properties* passed down by a parent component, and certain *state* defined in its constructor. The properties and state are intended to act as a *single source of truth*. Components also implement a **render** method, which returns the view described in an HTML-like language, *JSX*. The **render** method does not alter state directly, to separate the view generation from the component behaviour. Besides this, Components can have several methods which may hook into UI lifecycle methods, UI interaction events, and so on. These methods define the behaviour of the component.

More intricate UI design architectures were also considered, such as the *Model View View-Model* pattern. This pattern separates the data, UI, and behaviour into a model, view, and view-model respectively. The model and view are relatively straightforward, and the view-model maps the state stored in the model to information displayed in the view, and interaction events on the view to updates in the model. While this architecture is robust, it is relatively complex to implement, requiring at least two classes per UI view (the view and view-model). It also requires organising data into shared models, whereas the web client's different views will have relatively distinct sources of data - some will read from deployed smart contracts, others entirely from the UI. The increased complexity in implementing this kind of architecture was deemed unnecessary, as the typical React architecture is both sufficient and simple to write.

6.1.2 Web Client Components

Each screen of the application has its own component class, as well as some elements that are repeated throughout the web client on multiple views, and some that implement relatively complex behaviour. The main screens of the application are:

- The *connect* view, for connecting to a blockchain with an account. This is required so that financial smart contracts can be deployed to a blockchain, or interacted with if they are already deployed.
- The *main-menu* view, for navigation between views.
- The *composition* view, for composing and deploying financial smart contracts. Composed financial contracts can be evaluated here.
- The *monitoring* view, for viewing the state of deployed financial smart contracts and interacting with them. Deployed financial smart contracts can be evaluated here.

The *evaluation* view is also a major view, but it is shown in an overlay on the *composition* and *monitoring* views instead of having its own screen (to obtain a contract definition from these views). By selecting these as the main views of the web client, the major functionality which is required is neatly separated into one component per major function. The entire web client is always rendered

within a *main* view, which handles routing between the different views. The relationship between these views is depicted in figure 6.1.

Other views with their own components include a view for choosing an **or** combinator's sub-contract, setting an observable's value, acquiring an **anytime** contract's sub-combinator, and staking and withdrawing Ether - all on a deployed financial smart contract. There are also components for selecting a time for the insertion of a UNIX Epoch time into a combinator contract, inputting a deployed contract's address, and displaying instructions for writing a combinator contract. There are several smaller UI components as well, for displaying animations or specially-formatted information.

6.1.3 Other Functionality

Besides the UI functionality, there are several other facets of the web client's functionality. Static functions for interacting with the blockchain and deployed financial smart contracts were implemented in a **contract-utils** module. Evaluation of financial contracts written using the combinator DSL is implemented in the **Evaluator** class. This is represented as a class as it enables pre-processing and step-by-step evaluation of a financial smart contract, setting **or** combinator choices and acquisition times incrementally as they are encountered, which requires state to store pre-processing results and evaluation progress. The design and implementation of the **Evaluator** class is described in more detail in section 6.2.8.

6.2 Implementation

6.2.1 Technologies

As described earlier, the web client UI is implemented using *React*[24]. *Web3.js*[16], a JavaScript implementation of the Ethereum API, is used to interact with Ethereum blockchains. *Yarn* is used for package management. *webpack*[22] is used to run a development server, and build the production server files. *Babel*[2] is used to allow the usage of JavaScript ES6 functionality with incompatible packages, mainly for nicer module import syntax/functionality. *Sass*[4] is used to improve the structure of the UI styling, allowing the separation of styles into modules, and the definition of variables. *Browserify*[3] is used to allow node modules to run in the browser. *Moment*[32] is used to handle conversion between date/time formats. *Mocha*[21] is used as a testing framework, for automated unit and integration tests.

6.2.2 Contract Utility Functions

In order to handle interaction with the blockchain, the **contract-utils** module was created. This module has function implementations for setting up the Web3 connection, serializing/deserializing financial smart contract information, calling smart contract functions, and more. Several data classes are also defined, to represent smart contract interaction results. The exported functions are implemented as follows:

Financial Smart Contract ABI Functions

All of the functions on the financial smart contract ABI have a function implementation in the **contract-utils** module, in order to provide an abstraction of the smart contract. Each of these functions is asynchronous, returning a **Promise** which will either *resolve* or *reject* in the future. If an error occurs, the **Promise** is rejected with a descriptive error message, otherwise the **Promise** resolves with the correct return value. These functions are implemented as follows:

`getHolder`, `getCounterParty`, `getConcluded`, `getUseGas`, `getLastUpdated`, `getBalance`

These getter functions are easily handled by simply calling the respective method on the provided smart contract object from the given address, and resolving the returned `Promise` with the result.

`withdraw`, `stake`, `acquireContract`, `updateContract`, `setOrChoice`, `setObsValue`, `acquireSubContract`

Similar to the getter functions, these functions are implemented by calling the respective method on the given smart contract object from the given address, with all required parameters given through the function signature. The `Promise` is rejected if an error occurs, or resolved when the operation succeeds.

`getOrChoices`, `getObsEntries`, `getAcquisitionTime`

Similarly to the getter functions, these functions call the smart contract's respective method and await a result. Once a result is obtained, it must be deserialized before it can be returned. The implementation of the deserialization functions are described in section 4.3.3.

For `getOrChoices` and `getAcquisitionTimes`, the returned serialized vector of bytes is processed into an array of `Option` objects - a class implemented to represent Rust's `Option<T>` objects, with similar functionality.

The `getObsEntries` function obtains a list of observable entries from the contract method result - i.e. the arbiter addresses, values, and names of each observable - in a serialized form. An array of `ObservableEntry` objects is returned, a class that contains these three properties of an observable as well as its index. For each observable entry, the first 4 elements of the serialized vector are deserialized using the `deserializeAddress` method, an `Option` is deserialized from the following element(s), and then the name is deserialized from the following elements with `deserializeName`.

`getCombinatorContract`

`getCombinatorContract` similarly calls its respective method on the given smart contract, and deserializes the result into a human-readable combinator contract string. The serialized array is processed recursively; the first element being deserialized is converted into a combinator string, and then the following elements are deserialized based on the combinator's type.

`zero` and `one` can be returned immediately. `and`, `or`, and `then` all deserialize the following two sub-contracts, and then return their combinator joined with the two sub-contracts. `give`, `get`, and `anytime` can do the same, but for only one sub-contract. `truncate` takes the second element as a UNIX Epoch time and converts it into a human-readable time, followed by the deserialized sub-contract.

In the case of the `scale` combinator, the second element is used to indicate whether a constant or time-varying observable is present; if the value is constant it is simply added to the contract, otherwise the following elements are deserialized as an `ObservableEntry` (as for `getObsEntries`). Following this, the sub-contract is also deserialized.

Other Utility Functions

Besides the abstraction of the financial smart contract ABI, the `contract-utils` module provides some extra utility functions for dealing with blockchain interaction.

setupWeb3

This function initialises the *Web3* instance, which allows the web client to communicate with Ethereum blockchains. The function takes in a flag for whether or not to connect to *MetaMask*, a Google Chrome extension for managing blockchain accounts, as well as an optional blockchain URL. In order to initialise the Web3 instance, a provider must be obtained; the Web3 instance communicates with the blockchain through a provider. If using MetaMask, the Web3 provider is obtained from the browser window, otherwise an HTTP provider is initialised with the passed in blockchain URL. The Web3 instance is then initialised, or if already initialised its provider is reset.

unlockDefaultAccount

This function sets and permanently unlocks the default account for testing, using the Parity blockchain client's default testing account. This is only used for testing, *not* in production.

unlockAccount

This permanently unlocks the given address' account with a given password. This is used when providing a blockchain/account manually, which should only be used when security is not an issue. When security matters, MetaMask should instead be used.

loadAndDeployContract

This function deploys a financial smart contract to the connected blockchain. This is done by importing the financial smart contract code and ABI from another JavaScript module (set up as part of the smart contract build process), which are then used as parameters for a smart contract deployment transaction. The gas cost of deploying this smart contract is estimated and the smart contract transaction is deployed. The result of deployment is relayed asynchronously through a *Promise* object.

serializeCombinatorContract

This function is used to serialize a human-readable combinator contract, so that it can be passed across the financial smart contract ABI in the form of a vector of integers. It takes a combinator contract string and iteratively processes each combinator. For the majority of combinators, the combinator's numeric representation can simply be looked up and inserted into the serialized array, thanks to the simple compositional syntax of the DSL (see section [3.2](#)).

For the **truncate** combinator, the date needs to be serialized. If the date is given in UNIX Epoch format, then it can simply be verified and pushed onto the serialized array. If the date is given in a human-readable form, then it is parsed using the *Moment* package and converted into UNIX Epoch time.

For the **scale** combinator, the observable must be parsed. If a constant scale value is provided, the number 1 followed by the scale value is pushed. If a name and address are present, then the number 0 is pushed and the serialized address and name must be pushed onto the serialized array. The address is serialized by **serializeAddress**, which converts the address string into four signed 64-bit integers. The name is serialized by **serializeName**, which simply converts a string into an array of character codes by mapping, with the length of the name at the head.

verifyContract

The function **verifyContract** will take in a combinator contract string and verify that the syntax is correct. There is no semantic verification required, as the combinator DSL does not have any

semantic restrictions. `verifyContract` recursively verifies the contract syntax, returning an object containing either a `VerificationError` or the index at which the contract terminates. The `VerificationError` class allows the definition of an error with an error stack, to aid debugging in the contract composition view.

In the `verifyCombinator` helper function, the contract atoms are processed recursively. If a sub-contract returns an error, then the current atom's information is added to the error stack. `zero` and `one` can simply return the termination index. `give`, `get`, and `anytime` all return the verification of their sub-contracts, and `and`, `or`, and `then` similarly verify the two sub-contracts.

For the `truncate` combinator, the time must be verified. If the given time is not an integer, then it is verified as a human-readable date of the form '`<DD/MM/YYYY HH:mm:ss(ZZ)?>`', i.e. a date, time, and optional time zone. If the date format is incorrect, an error is thrown, otherwise it is converted into a UNIX Epoch time integer. If the time is an integer, it is verified to ensure that it fits within the bounds of an unsigned 32-bit integer.

The `scale` combinator can have a constant or time-varying scale value. If a constant scale value is found, it is checked that it fits within the bounds of a signed 64-bit integer. If a name and address are found, then the address is checked with `Web3.utils.isAddress`.

`isSmartContract`

This function takes an address and returns true if a smart contract exists with that address on the connected blockchain. To do this, the `Web3.eth.getCode` function is called to get the smart contract code associated with the given address. If any errors occur or no code is returned, the address is not a smart contract address.

`compareTime`

This is a standard comparison method which takes two times and returns 1, 0, or -1. `undefined` is treated as larger than any numerical value - this is because a financial contract with an undefined horizon is treated as if it expires later than a financial contract with a defined horizon.

6.2.3 Main View

The *main* component contains all other components rendered in the web client, and handles routing between them. Its state contains the Web3 instance, the current financial smart contract instance (if one exists), the current account address (if it exists), the application state, and an `Evaluator` object. The Web3 instance, smart contract instance, account address, and `Evaluator` instance are all passed down to the view currently being rendered. These views implement the behaviour which relies on these values.

The initial view is the blockchain connection view. A function is passed to the `ConnectAccount` component, which implements the connect view, to set the Web3 instance and account address. When this function is called, the main state is updated and the application state is changed to '`main-menu`'. This allows the application state to transition to the main-menu once the blockchain details are obtained.

The `MainMenu` component is given two functions as properties, one transitioning to the composition view, the other transitioning to the monitoring view. These functions update the main state to set the application state to the respective views when called, and are used to implement navigation

buttons.

Both the `Composition` and `Monitoring` components are rendered within a container, wrapped with a back button which updates the application state to `'main-menu'`. This allows navigation between these two views and the main-menu view.

6.2.4 Blockchain Connection View

The first major view a user reaches in the web client is the blockchain connection view, shown in figure 6.2. This view allows the user to connect to a blockchain via MetaMask, or by manually inputting the blockchain and account details.

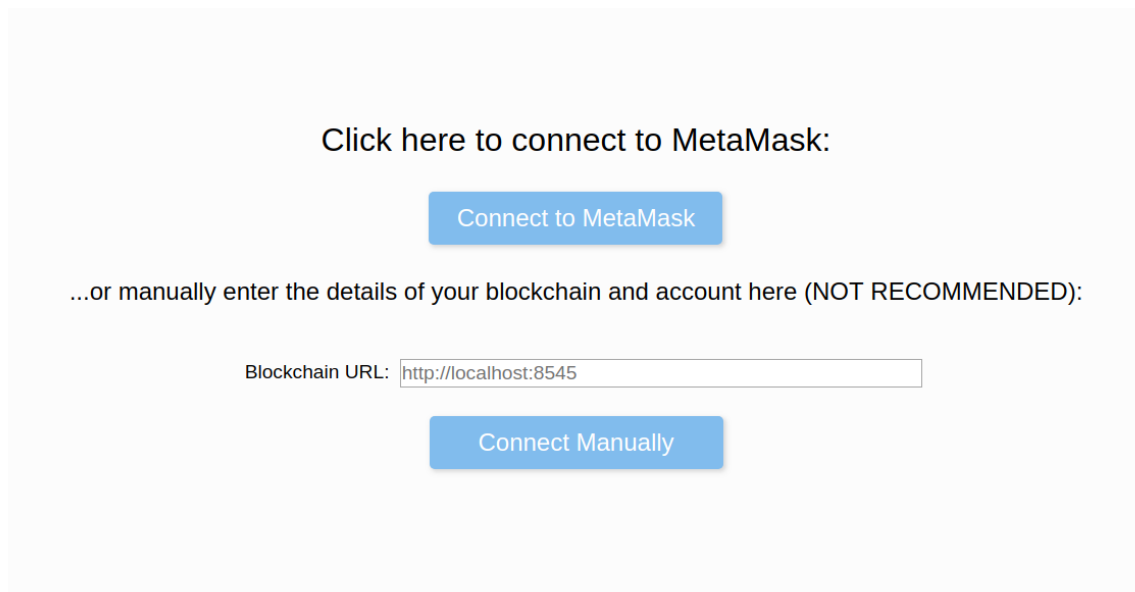


Figure 6.2: The blockchain connection view, used to connect to a blockchain.

This view is comprised of a button to connect to MetaMask, or several inputs for manually inputting blockchain data. When the MetaMask connection button is pressed, `setupWeb3` function from the `contract-utils` module is called with a MetaMask flag. This attempts to obtain the Web3 details from the browser; if successful, the application will update the main state and progress to the main-menu view, otherwise an error is displayed.

In order to manually connect to a blockchain, the blockchain URL must be input to the given input (defaulting to the typical local blockchain address) and a button must be pressed. This will also call `setupWeb3`, this time with the given blockchain URL. If the URL cannot be connected to, then an error is displayed, otherwise an address/password input is displayed. Once input, pressing the button will attempt to unlock the account with the `unlockAccount` function from `contract-utils`. If successful, the view will progress to the main-menu, otherwise an error is displayed.

While connecting to the blockchain, it is possible that silent failure will occur. To catch this, the `ConnectAccount` component will register a timeout on every asynchronous call to the blockchain for 10 seconds. If this timeout is reached, then an error is displayed and the connection is assumed to be unsuccessful. The Web3 instance is discarded, and the view reverts to its initial state (with extra error state).

6.2.5 Main-Menu View

The main-menu view is a simple view which displays two buttons, as shown in figure 6.3. Each button's click event handler is registered to a function passed down from the parent view as a property. One button navigates to the composition view, the other to the monitoring view.

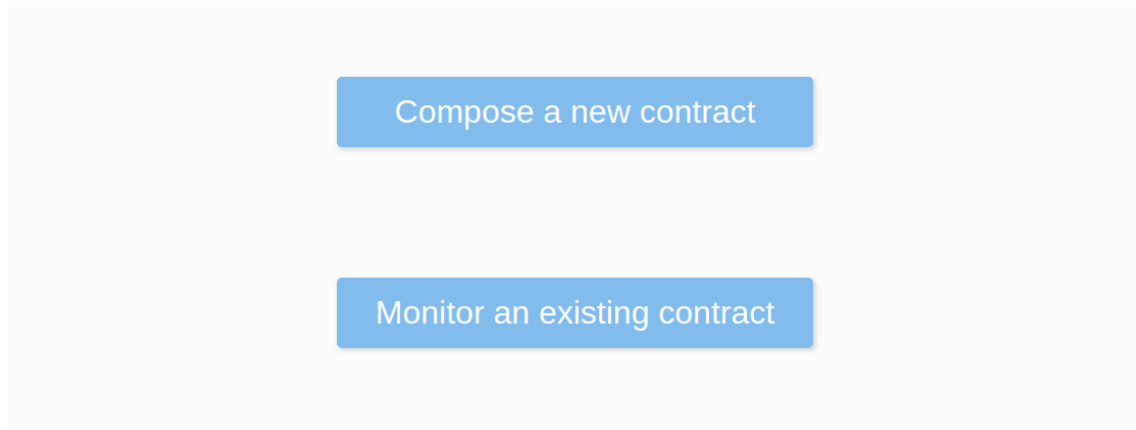


Figure 6.3: The main-menu view, used to navigate throughout the client.

6.2.6 Contract Composition View

The contract composition view displays a text-area input and some buttons, as shown in figure 6.4. The text-area input receives a combinator contract, and the option buttons allow the user to view combinator DSL instructions, insert a UNIX Epoch time into the text-area input, and evaluate or deploy the composed contract.

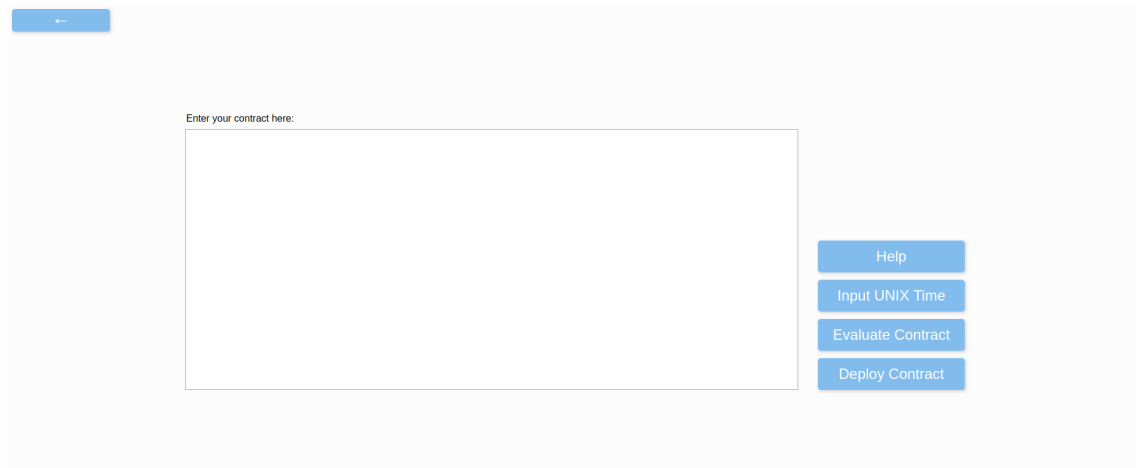


Figure 6.4: The composition view, used to compose SmartFin contracts and deploy their corresponding smart contracts.

The combinator contract is stored in the `Composition` component's state, and updates every time the text-area input's value changes. All of the buttons will open a *modal* component, i.e. a view displayed in an overlay over the current view. Whether or not each modal is visible is represented in the `Composition` component's state as a flag. Error messages/warnings are also stored in the state.

Upon pressing each button, the respective modal is opened. The modals contain a button to close themselves, which calls a `closeModal` function passed in as a property. The *help* modal is fairly

trivial, and the *evaluate* modal renders an `EvaluateControls` component described in section 6.2.8. The *input UNIX time* modal contains a `TimeSelect` component, which has a date and a time input, and sets state variables to reflect their value. The selected time is returned through a callback property. The time is then inserted into the text-area by getting the cursor location, splitting the contract at this location, inserting the time, and rejoining the contract.

The *deploy contract* modal contains a `DeployControls` component, which takes the combinator contract, the current address, a warning, and a callback for the deployed contract. Pressing the button to display the modal will first call `verifyContract`; if the contract is not valid, a descriptive error is displayed, otherwise the deploy modal is opened. The `DeployControls` component takes a holder address and a use-gas checkbox, and deploys the contract with the `loadAndDeployContract` function. The contract is returned with a callback property, which updates the main state with the deployed smart contract object.

6.2.7 Contract Monitoring View

The contract monitoring view is implemented in the `Monitoring` component, displaying a financial smart contract's state and allowing the user to interact with it, as shown in figure 6.5. When the view is entered, it will obtain the state of the financial smart contract instance currently stored in the main state, passed down as a property, using the getter functions from `contract-utils`. If no contract exists in state yet, an error is shown.

Contract deployed at: 0x5f3dba5e45909d1bf126aa0af0601b1a369dbfd7

You are the counter-party of this contract.

Contract Details		Load Contract
Holder:	0x057E231DaB35A789F5999056c8Ec775512609Cbb	Evaluate Contract Acquire Contract Stake Funds Withdraw Funds Update Contract Set Or Choices Set Observable Value Acquire Sub-contract
Counter-party:	0x004ec07d2329997267Ec62b4166639513386F32E	
Concluded:	false	
Acquisition Time:	None	
Holder Balance:	0	
Counter-party Balance:	0	
Uses Gas Upon Withdrawal:	false	
Last Updated At:	30/05/2019 14:51:01 +0000	
Combinator Contract		
one		
Or Choices		
This contract contains no or combinators.		
Observable Values		
This contract contains no observable values.		
Sub-contract Acquisition Times		
This contract has no anytime combinators.		

Figure 6.5: The monitoring view, used to monitor and interact with deployed financial smart contracts.

The user can press the *load contract* button to open a modal with a contract address input, which calls `getContractAtAddress` and returns the contract instance to the main state through a callback property. When the `Monitoring` view receives a new contract instance, it will attempt to read the state from the contract again. If any getter functions fail, an error is shown indicating that the smart contract may not be a compatible financial smart contract. Financial smart contract state is re-obtained every few seconds, registering a new timeout every time it's checked.

Besides displaying contract state, the **Monitoring** component has several buttons/modals for interacting with the financial smart contract. These all open modals containing components with simple inputs for the financial smart contract method parameters. These inputs are used to call the respective `contract-utils` method. The inner modal component will then call a callback provided by the **Monitoring** component which closes any open modal and re-obtains the financial smart contract state. There is also a button to open an *evaluate contract* modal, which is described in section 6.2.8.

6.2.8 Contract Evaluation

The *monitoring* and *composition* view both allow the user to open a modal to evaluate a combinator contract. This displays an **EvaluateControls** component, which lets the user step-through a financial smart contract and evaluate the final value, depicted in figure 6.6. While stepping through the contract, the user must provide acquisition times and choices for **or** combinators. The final value is represented as a sum of products of constant values and time-varying observable names, for more details see the user manual at appendix E.

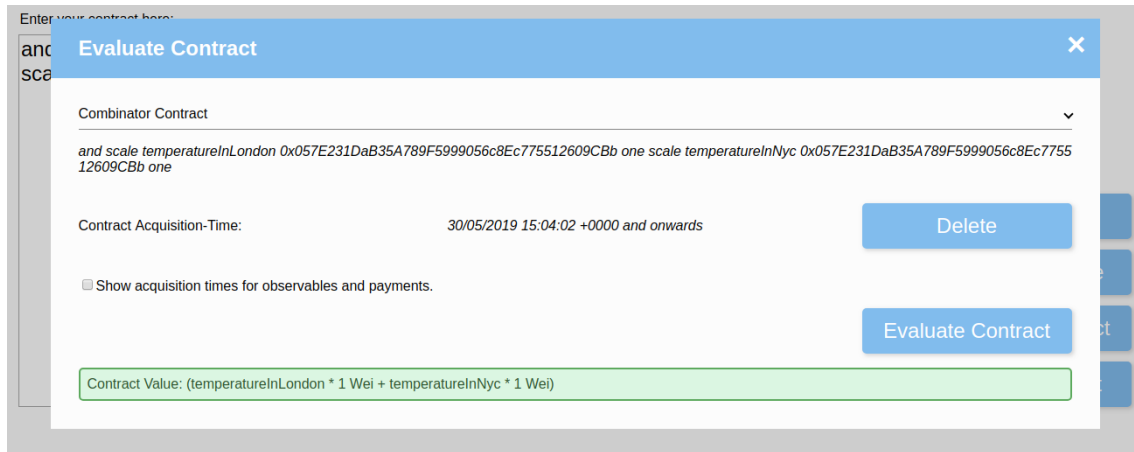


Figure 6.6: The *Evaluation* menu, after evaluating a SmartFin financial contract.

The Evaluator Class

An **Evaluator** class has been implemented order to maintain the state of step-through evaluation. This state is not just stored in the **EvaluateControls** component as it should persist across multiple views, and it keeps the **EvaluateControls** component mainly focused on the view. Furthermore, testing a pure JavaScript class is significantly easier than testing React components, and using automated testing for the **Evaluator** class is essential to thoroughly ensure that it operates correctly.

For context, the basic lifecycle of the **Evaluator** class is as follows:

1. A SmartFin contract is provided through the `setContract` method.
2. The contract is pre-processed with the `_processCombinators` method, to store state that will be useful for evaluation (like a map of sub-contracts to horizons).
3. The `hasNextStep` method is called to check if options must be set on the **Evaluator** before the SmartFin contract can be evaluated. If not, jump to step 7.

4. The `getNextStepThroughOptions` method is called to return the current set of options that the user must choose between and input while stepping through the contract (either an acquisition time or an `or` choice).
5. The `setStepThroughOption` method is called to input the chosen option.
6. The `hasNextStep` method is called, and if it returns true then return to step 4.
7. Once no more values must be input by the user for evaluation to be possible, `evaluate` is called to get the string representation of the value of the SmartFin contract with the current user input values.

In order to describe the implementation of the `Evaluator` class, it is first necessary to describe how acquisition times are represented in step-through evaluation of a combinator contract.

Representing Acquisition Times with Time Slices

When a combinator contract is acquired, the acquisition time will affect its value. For this reason, it is necessary to decide any relevant acquisition times in a combinator contract before it can be evaluated. One way to do this would be to have the user simply input any time, but it is possible to do better. In order to evaluate the contract we aren't really interested in *any* time, but only in several discrete segments of time in which the value of the contract is fixed - i.e. *time slices*.

For example, consider the contract `one`. Acquiring this contract at any time will result in the value 1 Wei being obtained, and so the acquisition time does not affect the value of the contract. This contract has *one* time slice. Now take the contract `truncate 10 one` - this contract has the value 1 Wei if acquired before the time 10, and 0 Wei if acquired after 10. As such, there are *two* time slices we are interested in, 0 to 10 and 11 to the end of time. As such, the user can evaluate the contract just by providing the time slices in which the contract will be acquired, instead of an exact time. This shows the user which distinct time periods are important, and allows acquisition times to be assumed when only one time slice exists.

In order to calculate the time slices in which a contract's value is fixed, the contract can be recursively evaluated. The combinators `zero` and `one` have a single time slice from 0 to the end of time - we can call this the *complete* time slice. Combinators like `give` and `scale` do not alter the time slices a contract has, only its value. For step-through evaluation, time-varying observables are simply represented as unknown variables, and so we can ignore the fact that a contract scaled by an observable doesn't have a fixed value in one time slice in the real world.

As shown previously, `truncate` will alter the sub-contract's time slices. Any `truncate` combinator acquired after its given horizon has value 0 Wei; as such, a time slice spanning the given horizon is split in two at the horizon, and any remaining time slices after this point are merged. However, if the `truncate` combinator's given horizon is later than the sub-contract's horizon, no change to the sub-contract's time slices occur. This is because the time slice following the sub-contract's horizon should be merged with value 0, and so splitting this will just result in two time slices with value 0.

The `get` combinator will merge any time slices before the horizon. This should result in only two time slices, as acquisition after the horizon will always have value 0 and thus should be represented by a single time slice.

The `and` and `or` combinators will merge the time slice sets of the two sub-contracts. In this context, merging the sets of time slices means that the resulting set of time slices has boundaries at every boundary in both sets of time slices. For example, the contract `and truncate 10 one truncate 20 one` has time slices 0 to 10, 11 to 20, and 21 to the end of time. The two sub-contracts' time slice sets are 0 to 10 and 11 to the end of time, and 0 to 20 and 21 to the end of time. The `then` combinator has the time slice set of the first sub-combinator, with all of the time slice boundaries

from the second sub-combinator's time slice set beyond the first sub-combinator's horizon added.

The `anytime` operator does not change the sub-combinator's time slices, but it does allow a time slice to actually be acquired. This will be another place where a user can choose an acquisition time slice during step-through evaluation besides the top-level contract acquisition time.

Evaluation Pre-processing

When a contract is supplied to an `Evaluator` instance, it is pre-processed by the `_process-Combinators` method. This method recursively processes the combinator contract, returning a `ProcessResult` object. This object has a horizon, a set of time slices, and a list of the time slice sets of each `anytime` combinator. The purpose of pre-processing the combinator contract is to obtain a mapping of combinators' indexes to their sub-contract termination indexes and horizons, and set of time slices for each `anytime` combinator and the top-level contract.

When the combinators `zero` or `one` are encountered, the combinator to sub-contract termination index mapping is updated to signal that the sub-contract terminates at the current index. This information is stored in a `NextMap`, a class implemented which takes an index and returns the value stored at the closest greater-or-equal index (if one exist). This is because there is no need to store the index that a sub-contract terminates at for every combinator, only for those that alter the flow of the contract - like `zero` or `one` that terminate a sub-contract, or `and` which combines two sub-contracts. The `ProcessResult` is returned with an undefined horizon, and empty time slice sets.

When the `truncate` combinator is encountered, the time is parsed. If the horizon is greater than the sub-contract's horizon, that horizon is taken instead. The sub-contract's set of overall contract time slices is then modified as described in section 6.2.8, and the horizon is added to the mapping of combinators to horizons (also a `NextMap`). The `ProcessResult` of the sub-contract is returned with the new horizon and time slices set.

The `give` and `scale` combinators do not modify the `ProcessResult` of their sub-combinators. The `get` combinator will merge the sub-contract's time slices before the horizon, as described in section 6.2.8. The `anytime` combinator will add the sub-contract's overall time slices set to the array of anytime time slice sets, and return the updated result.

The `and`, `or`, and `then` combinators all update the combinator to sub-contract termination index map, as they alter the contract flow - combinators before them are part of a contract which terminates after both sub-contracts, not just the first termination that occurs. The time slice sets are also modified as described in section 6.2.8 - `and` and `or` merge the two sub-contracts' time slice sets, and `then` adds the boundaries from beyond the first sub-contract's horizon in the second time slice set to the first. The horizon map is also updated for all of these combinators to the latest of the two sub-contracts' horizons.

After pre-processing the contract, the `Evaluator` instance is ready to begin stepping through the contract and setting options.

Getting a Set of Step-Through Options

The function `getNextStepThroughOptions` returns a set of options when the step-through evaluation has reached a point where user input is required, i.e. there are several possible acquisition time slices or `or` combinator choices. The options available at that point in the contract are then returned to the caller. If the `includePast` flag is set to false, then time slice options before the current time will not be shown.

If the contract is at the beginning of step-through evaluation, the top-level acquisition time must be set. This must be one of the top-level time slices obtained by pre-processing the combinator contract, and so this set is returned (including the slice after the horizon). If the `includePast` flag is false, the time slices set is split at the current time, and time slices before this are not returned.

If the contract acquisition time has been set and options still need to be set, then the current combinator in step-through evaluation will either be an `or` combinator with two non-expired sub-contracts, or an `anytime` combinator. For an `or` combinator, the options true and false are returned. For an `anytime` combinator, the set of time slices for that `anytime` combinator obtained through pre-processing are returned, with the slice after the horizon removed. The time slice set is split at the current step-through time slice, and the time slices before this point are not returned.

After a set of options has been returned, the chosen option can be set by calling `setStepThroughOption`.

Setting Step-Through Options

The function `setStepThroughOption` will take an option obtained from the `getNextStepThroughOptions` function and push it to a step-through values array. These values are used to evaluate the final value of the contract. If the option is an acquisition time slice, then the current step-through time slice is set to that time slice. `anytime` combinators encountered while stepping through the contract will not return time slices before the current step-through time slice, as you cannot acquire an `anytime` combinator before its parent. If an `or` combinator is encountered, then the combinator index - representing the location of the contract step-through - is moved to the point of the correct sub-contract. This uses the combinator to sub-contract termination index map.

After setting a step-through option, the method `_goToNextStep` is called. This steps through the contract until another combinator requiring human input is encountered. This is done in a while loop, by passing through the contract while keeping track of the step-through time. If an `or` combinator is met with an expired sub-contract, the function steps through the non-expired sub-contract. If an `anytime` combinator is met with only one possible acquisition time slice, it is chosen automatically. If a `get` combinator with no horizon is met or the horizon of the encountered combinator has passed, it is treated as the end of a sub-contract. If an `and` combinator is encountered, it is recorded in a stack. When the end of a sub-contract is reached, the stack of `and` combinators is popped to visit the second sub-combinator. If the stack is empty, then stepping through is complete.

The `deleteStepThroughOption` allows step through options to be removed. This is implemented by simply deleting any stored step-through options for combinators beyond the given combinator index, resetting the current step-through time and position to that of the combinator index, and removing values from the `and` combinator stack.

Evaluating a Contract Step-Through

Once all step-through options have been set, the contract is ready to be evaluated. Calling the `evaluate` method will call the `_stepThroughEvaluate` function, with the top-level acquisition time and a copy of the step-through options array.

`_stepThroughEvaluate` recursively evaluates the combinator contract with the given step-through options, returning a `StepThroughEvaluationResult` object. The `StepThroughEvaluationResult` class keeps track of a stack of intermediate results. These intermediate results represent payments by the `one` combinator, scaling by a constant value or a time-varying value by the `scale` combinator, and addition by the `and` operator. They also contain a time slice, which can be used to show the final value with time slices for observables and payments. As the contract is evaluated

bottom-up, the evaluation result is scaled and added by earlier combinators. The result is then formatted in a human-readable form by consuming the stack.

Back to the `_stepThroughEvaluate` function - this function evaluates the contract value from the bottom up, keeping track of a `StepThroughEvaluationResult` object. If the sub-contract being evaluated has expired, a `StepThroughEvaluationResult` with value 0 is returned. If `zero` or `one` is encountered, a new result is returned with the correct value and time slice. If `give` is encountered, the sub-contract result is scaled by -1.

Encountering `truncate` will simply return the sub-contract, as the horizon is always checked. `scale` will cause the sub-contract result to be scaled by a scalar, or by a named observable. `get` will return the sub-contract result obtained by evaluating at the horizon, and `anytime` will return the result obtained by evaluating with the chosen time slice from stepping through the contract.

Encountering `and` will cause the results of the two sub-contracts to be added, `or` will return the result of the chosen sub-contract, and `then` will return the result of the correct sub-contract at the given time.

The final `StepThroughEvaluationResult` will contain a stack of intermediate results. When processing this into a string (starting from the top), encountering a `scale` by constant value will pass the scalar down to the evaluation of the next intermediate result, multiplied with the current scalar (acting as an accumulator). Encountering a `payment` simply returns the payment value multiplied by the given scalar as a string. Scaling by an observable name will return the string '`<observable name> *` ' followed by the next intermediate result's value string. An addition result (from an `and` combinator) returns the string value of the next intermediate result, and then the following result (after the intermediate result is consumed up to a payment result), bracketed with a plus symbol in between the two. The result is a human-readable value representing the contract's value with the given options. The times of payments and observable value acquisitions can also be displayed, to make real value estimation easier.

The EvaluationControls UI Component

The `EvaluationControls` component, shown in figure 6.6, takes the given combinator contract and sets it on the `Evaluator`. It then simply shows the options returned by `getNextStepThroughOptions` in a drop-down select input. Once an option is chosen, it is set with the `setStepThroughOption` function. Previous chosen values are displayed, obtained from the `Evaluator` object by a getter. Once all options are set, a button to evaluate the contract can be pressed. This calls the `evaluate` method, and the result is displayed on the view. This follows the basic lifecycle of the `Evaluator` function closely.

6.3 Evaluation

In order to evaluate the web client, the architecture and functionality of the web client can be qualitatively evaluated. The use of automated testing to ensure correctness can also be evaluated.

6.3.1 Web Client Design

The general architecture of the UI components was fairly effective. Smaller UI views that were re-used across the application, such as the `Modal` component, were extracted into their own class; this allows for comprehensive code reuse. The separation of each view into its own class, including views displayed in modals, helped to prevent complicated and messy component classes.

One issue with the architecture of the web client is that the UI components are very difficult to test programmatically. This is because every component had all of its logic implemented as methods on the component, which are typically called by UI events. Instantiating these components and measuring the state changes through method calls is difficult in their current form. One alternative design pattern considered was the *Model View View-Model* pattern, where all logic for each component would be encapsulated in a view-model class. This would be a lot easier to test, although due to the small/simple nature of the web client it was not implemented as it would require significantly more effort. If the web client were to be extended with more complex UI behaviour, then the current architecture would definitely be a hindrance, but it is more acceptable given the relative simplicity of the current UI of the web client.

A compromise that was made with the web client is that it must be run from a server (locally or remotely) and opened in Google Chrome. Ideally, the client would instead run in its own application instead of requiring a server to be run and navigated to. An alternative that was seriously considered was deploying the client as an Electron application, effectively still being a web client but running in a separate program, instead of being served to a browser. Despite the benefits to alternative approaches like this, the need for MetaMask support is very important due to the security benefits it brings, and as such the only viable approach is to serve the web client to a Google Chrome instance.

6.3.2 Web Client Functionality

The web client definitely improves the ease and simplicity of deploying a financial contract as a smart contract. The basic functionality of composing and deploying a financial smart contract is implemented, with a relatively simple UI that makes inputting all of the required information easy and intuitive. There is also a *help* screen, with full instructions of how to compose a financial smart contract. If any user has difficulties using the web client, there is also a full user manual detailing every feature in the web client, and describing the combinator DSL in detail.

The verification and error-reporting of the composition view is another feature which makes it easier to develop financial smart contracts; any errors in the combinator contract will be reported with a detailed description, and a basic error trace to aid with locating the error. The evaluation view also allows a user to step through a contract, ensuring that their implementation is correct.

One feature that would be very useful for estimating the risk of a financial smart contract would be the ability to find the maximal value of the contract, instead of just stepping through to find a value with a specific set of inputs. A major roadblock in implementing this was observables; as far as the application is aware, every observable is an unknown which could take any value. Without some way of modelling the value of an observable, it is impossible to estimate whether one contract involving observables has a greater maximal value than another, and probabilistic numerical modelling of variable values is a whole research field of its own right.

One possible approach that could have been implemented was to allow the maximal evaluation of any contracts without observables. While it would definitely be possible to add this functionality to the `Evaluator` class, it has not been prioritised as the maximal value of contracts which don't contain observables are typically not hard to evaluate mentally - or at least to estimate the order of magnitude of their value.

Another possible option was allowing a user to provide the value of observables, but given their variation over time this would be very difficult to implement from a UI standpoint; the value of all observables over all time periods would be required in order to find a contract's maximal value, which is a huge amount of information to manually input. Allowing the user to input data with which the values of observables could be modelled over time would be feasible in terms of UI, but the implementation of numerical modelling in the web client would be very difficult.

A third option could be to allow the counter-party to define certain bounds on observables when authoring the contract. This could allow evaluation to make assumptions with regards to the values of observables, and estimate a maximal value. While this would be useful for observables with low variation over time, more volatile observables would still be difficult to represent in a useful way with this system. Furthermore, this system could potentially be quite difficult to implement correctly, although a basic implementation may be feasible.

Due to the issues present with each potential solution, or the high development effort required (along with time constraints of the project), maximal evaluation was not implemented.

6.3.3 Evaluator Design

From the beginning, the **Evaluator** class was designed with step-through evaluation across multiple views of the web client in mind. This means that pre-processing was a useful tool, as one combinator contract may be re-evaluated with various different inputs. For an example of the utility of pre-processing, if the terminating indexes of sub-contracts were not pre-processed then they would need to be calculated every time they are needed, like every time an **or** branch is skipped or an **and** combinator is visited. This may be more efficient in the short term for certain contracts, but if many evaluation paths are calculated for one contract then pre-processing will be significantly more efficient in the long term.

One thing that is inefficient about the **Evaluator** class is that it uses about four passes to evaluate one contract - one pre-processing pass, one pass to step-through and set options, one pass to evaluate with the options, and one (albeit smaller) pass to format the evaluation results. If we accept the use of pre-processing as a benefit as discussed already, it still uses three passes.

The reason for this is that the originally the step-through of the contract was implemented completely separately to the evaluation, in only two passes. The stepping-through of the contract happens in an iterative, top-down manner; this is because options must be filled in as they are encountered, and it's impossible to know which options will be encountered unless earlier options are filled in. Evaluation, on the other hand, occurs recursively in a bottom-up manner; **zero** and **one** combinator results are scaled and summed on the recursive path up the contract. Originally, the formatting of the string value of the combinator compounded itself with scalar and variable factors and sums on the recursive return up the contract. This was very complicated and error-prone, however, and was replaced with a stack-based system of constructing the final value string, thus adding the third evaluation pass.

It is possible that the intermediate evaluation result stack could actually be constructed as a queue during the contract step-through, completely skipping the separate recursive evaluation step. This would likely require the **Evaluator** class to be heavily re-implemented however, to prevent the already complex step-through methods from becoming unwieldy. Unfortunately, this was not implemented due to time constraints, but it likely could cut the total number of passes down from 4 to 2-3. This is not too much of an issue as the order of magnitude of complexity is linear either way, and any speedup would be indistinguishably better for non-extreme use-cases, but it *would* be a more efficient approach.

6.3.4 Testing

Unit tests were implemented to thoroughly test the **Evaluator** class, and the **contract-utils** functions, as well as several of the helper classes like **NextMap** and **TimeSlices**. Most of the UI testing was done manually, as almost all of the UI behaviour handled either updating the visual UI or passing data to/from the **Evaluator** class or **contract-utils** functions. As such, automating testing of the more complex facets of the application was deemed more important. Automated testing was implemented with the *Mocha* framework.

`contract-utils`

The `contract-utils` tests were separated into several sections: utility function tests, serialization/deserialization tests, verification tests, and contract interaction tests. In total, there are 59 unit tests for the `contract-utils` module, made up of 450 LoC.

Utility function tests cover functions for serializing/deserializing specific parameters - like observable names or addresses - validating values, or checking/getting a smart contract at an address. The small serialization/deserialization functions are each checked in pairs, to ensure that the original value remains when serialized and then deserialized. The validation functions are tested with all types of valid and invalid inputs. The smart contract getting function was tested with a deployed financial smart contract, and the smart contract checking function was tested with an account address and a deployed smart contract address.

The bigger serialization/deserialization tests ensured that combinator contracts are correctly serialized and deserialized. The `serializeCombinatorContract` method is tested with combinator contracts for all combinators, and all parameter types for each combinator, by manually creating a correct serialized version and checking the function output. To test the `deserializeCombinatorContract` method, combinator contracts are created for all combinators and then serialized with the `serializeCombinatorContract` method. The deserialization of the serialized version is then checked against the original contract definition.

The contract verification tests ensure that a valid contract with all combinators is verified correctly, and that all possible types of syntax error are caught by the verifier - these include missing combinators, incorrectly-spelled combinators, incorrect parameter types/values, invalid addresses, etc.

The contract interaction tests test each function for interacting with a financial smart contract via the ABI by deploying a financial smart contract and calling the function on it. This is more of an integration test to ensure that the communication between the web client and financial smart contract are set up correctly, as the financial smart contract methods are unit tested in the Rust implementation.

Evaluator

The `Evaluator` class is thoroughly tested with unit tests - there are 51 unit tests in total, made up of 408 LoC. The calculation of the overall contract's horizon (through pre-processing) is tested with all combinator types. The calculation of time slices is tested with all combinator types, against the specification laid out in section 6.2.8. The step-through evaluation options for all option types are tested, and automatic completion of options is also tested in every context - i.e. expired or sub-contract, or single `anytime` acquisition time slice. `hasNextStep` is also tested with completed contracts.

`deleteStepThroughOption` is tested by deleting an already set option, and checking the set of options offered. The value string resulting from evaluation of contracts with every kind of combinator, with all of their parameter types, is also tested - with step through options chosen in the tests.

Other Tests

Besides the tests for the larger modules, there are also tests for the `TimeSlices` and `NextMap` modules. There are 10 unit tests for these modules in total, made up of 87 LoC.

Tests for the `TimeSlices` module test every method on the class with a set of time slices, to ensure that the operations described in section 6.2.8 - i.e. merging sets of slices, merging slices before/after a point in time, etc - all operate as described. All of these tests pass.

Tests for the `NextMap` module also test every method on the class, to ensure that the correct value is returned based on the passed-in key - i.e. that the next greater-or-equal key's value is returned, and that the keys are sorted correctly (*not* lexicographically).

6.3.5 Conclusion

Overall, thorough unit testing of the contract interaction methods and the `Evaluator` class support their correctness. Automated testing was lacking for the UI components however, due to the difficulty of testing React components; a different architecture could have made testing easier, but required a lot more work to implement. As such, manual testing is relied on instead.

The design of the web client as a whole is effective at reducing code re-use, and encapsulating different parts of the client's functionality. All necessary functionality is implemented, including composition, deployment, and interaction with financial smart contracts. Basic step-through evaluation of combinator contracts is implemented, but its utility has room for improvement through implementing maximal evaluation (albeit after significant effort), and its efficiency could be improved.

6.4 Remarks

At this chapter's conclusion, the design and implementation of a financial smart contract and web client to compose, deploy, monitor, and evaluate it has been described. The result is a system which easily allows a user to access this functionality, with minimal technical skills required.

The next chapter will provide some evaluation of the final product as a whole, and the benefits that the contributions made can bring to the worlds of financial and smart contracts.

Chapter 7

Evaluation

Due to the somewhat segmented nature of this project, the implementation-based chapters so far have each included an evaluation. In this chapter, these individual evaluations will be brought together alongside new analyses to evaluate the contributions made in this project holistically.

7.1 Individual Chapter Evaluations

Each implementation-based chapter in the main body has its own evaluation section, as these chapters are relatively distinct in content. These sections are as follows:

- Evaluation of the top-level smart contract implementation of SmartFin, in section [4.4](#).
- Evaluation of the SmartFin combinators' semantics implementation in the smart contract, in section [5.3](#).
- Evaluation of the web client, in section [6.3](#).

Overall, based on the evaluations of each of these sections, the smart contract implementation of SmartFin provides sufficient functionality to represent a SmartFin financial contract in the form of a smart contract. Certain compromises had to be made, such as the use of retroactive payments instead of scheduled payments, and the provision of observable values by an external user as opposed to obtaining them dynamically; these compromises are unavoidable when writing smart contracts on the Ethereum platform, however, and they do not significantly impact the functionality of the smart contract implementation. The representation of SmartFin contracts through the implemented smart contract was thoroughly tested to ensure correctness, providing some guarantee towards the behaviour of these financial smart contracts.

The web client makes the composition, evaluation, deployment, and monitoring of financial smart contracts quick and easy. This makes it easier to create correct smart contract implementations of SmartFin financial contracts, and mitigates the risk of errors in their composition. Contract evaluation could be expanded upon by implementing maximal/minimal evaluation of SmartFin contracts, but this would likely require some level of numerical modelling to handle observables - unfortunately, this is outside of the scope of this project. The web client was also thoroughly tested to ensure the correctness of functionality dealing with the blockchain and SmartFin contract evaluation; UI testing could have been improved, however, but this does not impact the essential functionality of the web client greatly.

7.2 User Feedback

In order to further evaluate the final product of this project (consisting of the smart contract implementation of SmartFin and the web client) as a whole with reduced bias, several people experienced with blockchain technology were interviewed for their feedback. The interview consisted of running through the web client, explaining the SmartFin DSL, and demonstrating the behaviour of several financial smart contracts. Some Solidity implementations of financial smart contracts were also shown to the interviewees for comparison, which are listed in appendices [B](#), [C](#), and [D](#).

The SmartFin DSL was explained to the interviewees, as well as the method of producing SmartFin financial smart contracts - smart contract instances which represent a SmartFin financial contract - by providing SmartFin contracts to a smart contract's constructor. Several SmartFin financial contracts were explained and demonstrated to the interviewees, along with Solidity implementations of these contracts (listed in appendices [B](#), [C](#), and [D](#)). The most obvious difference noted by the interviewees between the sets of contracts was succinctness - as discussed in section [7.3](#). They noted that writing the SmartFin contracts would be significantly less work, and the simplicity of SmartFin was praised as it makes SmartFin contracts significantly less error-prone - whereas Solidity contracts need to be written carefully to prevent critical bugs like reentrancy errors and integer overflow/underflow. It was mentioned that those writing financial contracts would likely prefer a more rigid implementation language for the guarantees of correctness it provides, rather than a less-restricted but complex language like Solidity. This shows the utility of using SmartFin to create financial smart contracts over a high-level language like Solidity.

The operation of financial smart contracts was also explained and demonstrated to the interviewees, along with the concessions made due to the limitations of Ethereum smart contracts - such as the requirement of an `update` method to deal with contracts over time, and the provision of observable values from arbiters instead of obtaining them dynamically. All who were interviewed understood these limitations, and no alternative approaches were deemed better than those implemented. The compromises made were accepted by the interviewees, who agreed that they did not greatly harm the utility of the final product. As such, this supports the idea that the smart contract implementation of SmartFin is fit for purpose.

The interviewees had a brief discussion on the use cases of financial smart contracts. It was agreed that smart contract representations of financial contracts would be useful in/between financial institutions, for simplifying payments required by financial contracts the company is involved in, or for auditing purposes. The use case of an individual deploying financial smart contracts on a public blockchain was also considered, and several issues were brought up. As mentioned in section [4.4](#), there is no way to bind users to the obligations laid out in a financial smart contract. As such, financial smart contracts between two users on a public blockchain are less useful as it is difficult for the two parties to use some other form of binding (like a traditional legal contract). Furthermore, it was noted that typical members of the public do not often engage with financial contracts without the presence of a party with relevant legal knowledge, making them unlikely to create their own financial smart contracts. Besides these points, the lack of privacy on a public blockchain was also brought up as a deterrent; a private blockchain can typically only be accessed by trusted parties, but a public blockchain can be viewed by anyone. This would allow financial smart contracts to be visible to anyone on the public blockchain, which could be off-putting due to the lack of privacy. Because of these points, the usage of financial smart contracts by individuals on public blockchains was deemed as an unlikely scenario, and thus prioritising consideration of private blockchain usage during implementation may have been the correct approach.

One interesting point raised by the interviewees regarded how the gas costs of financial smart contracts (i.e. the fees required for deployment and execution on the blockchain) affect their value. The web client allows SmartFin contracts to be evaluated for their cost/value, but these evaluations do not take gas costs into accounts. In situations where SmartFin contracts deal with a very small amount of Ether, it may be the case that gas costs greatly alter the resulting value of a contract. For example, when deploying the contract `one`, the gas costs are likely to be an order

of magnitude greater than the evaluated value of the contract, thus greatly affecting the actual final cost/value of the contract (see section 7.3 for gas cost analyses). As such, some method of taking gas costs into account during evaluation could be useful, or at least the ability to preview gas costs of deployment (as deployment is typically the most expensive operation in terms of gas). This issue is more important for public blockchains where miners only operate because of gas fees, as private blockchains are less likely to require/rely on expensive gas fees where the miners will instead be operating for the sole purpose of updating the blockchain; as such this suggestion is not an essential feature for the web client, but could be implemented in the future relatively easily in a basic form.

In summary, the interviewees found that the creation of SmartFin financial smart contracts through the web client required much less development effort than Solidity representations of equivalent contracts, and that using SmartFin was likely to result in a far lower risk of erroneous behaviour than Solidity. Furthermore, the behaviour of SmartFin’s smart contract implementation was found to be fit for purpose, and its compromises acceptable. The use case of financial institutions deploying financial smart contracts to a private blockchain was deemed significantly more likely than individuals deploying them to public blockchains, for trust and privacy reasons. Additionally, the gas costs of deploying/interacting with financial smart contracts could potentially be considered when evaluating these contracts, although this is less useful on a private blockchain.

7.3 Solidity Case Studies

For a user intending to implement financial contracts as smart contracts, the other main option besides implementing it in SmartFin and deploying a financial smart contract is to deploy a bespoke smart contract written in a smart contract language which represents the financial contract. In order to compare the use of SmartFin in the web client to produce financial smart contracts with the implementation of bespoke financial smart contracts, several SmartFin examples have been implemented as Solidity smart contracts to compare the benefits and weaknesses of each approach. For clarity, any financial smart contracts deployed by the web client will be referred to as a *SmartFin smart contract*, whereas any Solidity implementations of SmartFin contracts will be referred to as a *Solidity smart contract*.

Three SmartFin contracts have been implemented as Solidity smart contracts; these contracts span every combinator in the DSL, in order to provide some level of comparison for all combinators. The more complicated SmartFin contract examples are also used/explained in section 2.3.3 (in the original DSL syntax, with the equivalent SmartFin behaviour). Each SmartFin smart contract and Solidity smart contract are compared by ease of implementation, accuracy of implementation, and efficiency/gas costs (i.e. the fees required for deployment/execution on the blockchain). For context with regards to gas estimation, 10^{18} Wei is equivalent to 1 Eth, which is equivalent to \$262.25 USD at the time of writing[5].

7.3.1 Simple Contract

The *simple contract* example is defined as follows:

```
one
```

This contract was used in order to compare the general operation of SmartFin smart contracts with Solidity smart contracts, without comparing complicated combinator behaviour. The Solidity implementation of the simple contract is listed in appendix B.

Ease of Implementation

For the *simple contract*, the implementation as a SmartFin smart contract using the web client is trivial. Just inputting the contract `one` with a holder is all that’s required. As such, the SmartFin

smart contract is very easy to compose and deploy, and the risk of introducing errors is very low.

The Solidity implementation of the *simple contract*, however, is anything *but* trivial. Overall, the Solidity smart contract takes 140 lines to represent the *simple contract* (or 100 without comments). This is because it needs to define various methods that the user can interact through, just as the SmartFin smart contracts have a pre-defined ABI. This highlights the main issue with writing Solidity implementations of financial contracts; when writing *any* Solidity smart contract implementation of a financial contract, all of the typical boilerplate methods for handling payments, staking/withdrawing funds, and so on must be written from scratch. The comparative benefit to deploying SmartFin smart contracts from the web client is that only the SmartFin contract definition must be written by the contract author, and *not* the smart contract implementation of the SmartFin contract, thus making the barrier to entry for a user significantly lower.

A notable issue with implementing the *simple contract* in Solidity is how much work has to be done to avoid introducing errors. The functions `safeAddSigned` and `safeSubSigned` have been implemented to prevent overflow or underflow from occurring when adding to or subtracting from a party's balance, as by default this does not throw an error. Additionally, the `withdraw` function can be at risk of a reentrancy attack - if the balance is decremented *after* the Ether is transferred to the caller, then the contract can be entered again during the withdrawal transaction to repeatedly withdraw funds from the contract. This demonstrates the risks that exist when writing Solidity smart contracts which would be avoided by creating SmartFin smart contracts, which have been thoroughly tested. This is another benefit of using SmartFin smart contracts over implementing bespoke Solidity smart contracts.

Accuracy

The SmartFin smart contract and the Solidity smart contract effectively behave the same, so the accuracy with which both smart contracts represent the *simple contract* is equivalent. Implementing the Solidity smart contract does *not* allow the compromises made in the SmartFin smart contract implementation to be avoided (like retroactive payment) as these issues stem from the nature of the Ethereum platform, which both smart contracts are hosted on. The only benefits the Solidity smart contract has are the lack of superfluous ABI methods, although this does not effect the behaviour of the smart contract anyway and so this is only a superficial benefit.

Efficiency and Gas Costs

In order to evaluate the efficiency of the two smart contract implementations of the *simple contract*, the gas costs of executing certain methods on the smart contract instances were measured and recorded in table 7.1. Gas costs are the fees associated with deployment/execution on the Ethereum blockchain, used to reimburse the miner that computes this execution.

Gas Cost for Method Calls in Wei - <i>simple contract</i>			
Method	SmartFin S.C.	Solidity S.C.	SmartFin to Solidity Ratio (3 s.f.)
constructor	4875700	1770016	2.75
acquire	180417	148682	1.21

Table 7.1: The gas costs of certain method calls on the *simple contract*'s SmartFin and Solidity smart contract instances.

As shown in table 7.1, the constructor for the SmartFin smart contract instance is 2.75 times more expensive in terms of gas cost than the constructor for the Solidity smart contract instance. The reason for this is likely the deserialization that the SmartFin smart contract must carry out to create the combinators. The SmartFin smart contract also stores some extra state in the form of empty vectors for things like `or` combinator choices and `anytime` sub-contract acquisition times, which would cost extra gas. The SmartFin smart contract also contains more code than the Solidity smart contract as it contains logic for handling all combinators, including those not used

in the *simple contract*. The Solidity smart contract, on the other hand, does not need to process a serialized SmartFin contract definition as its implementation is bespoke to the *simple contract*, and does not need to store superfluous unused data.

The `acquire` method is closer in gas cost for the two smart contract implementations, as the SmartFin smart contract's `acquire` method call only costs 1.21 times as much as the Solidity smart contract's `acquire` method. This is likely because the *simple contract* only involves the `one` combinator, and thus both smart contracts have to do very little work to update the smart contract state to represent the payment occurring. It is important to note that the `acquire` method also calls the `update` method in the SmartFin smart contract, so acquiring the contract also updates it.

Overall, this comparison shows that the Solidity smart contract implementation for the *simple contract* is significantly more efficient than the SmartFin smart contract implementation, due to the generic nature of the SmartFin smart contract compared to the bespoke nature of the Solidity smart contract. The `acquire` methods do not differ much, as this contract only has a small effect upon acquisition.

7.3.2 European Option

The *European option* example contract is defined as follows:

```
get truncate <01/01/2020, 00:00:00> or
  scale 500 one
  zero
```

This contract represents a *European option* for a contract where the counter-party pays the holder 500 Wei. This contract is used to allow the comparison of a more complicated financial smart contract with many recursive combinators and multiple options, to the bespoke implementation of a bespoke Solidity equivalent. The Solidity implementation of the *European option* is listed in appendix C.

Ease of Implementation

Implementing the SmartFin contract representing the *European option* is relatively straightforward. It is clear that the contract has an effect at a certain time, thus a `get truncate` combination is used to represent this. The choice of whether to acquire a sub-contract `c` can be represented easily by `or c zero`, and the contract for a counter-party to pay the holder 500 Wei is trivial. The total number of combinators required is 6. The Solidity smart contract implementation of the *European option* contract requires almost 200 lines, which again takes significantly more development effort than the SmartFin smart contract.

One interesting note is that much of the Solidity implementation boilerplate can be re-used from the *simple contract* implementation, mainly requiring different behaviour upon acquisition to represent the combinators. The re-use of this code could allow for some time-saving when implementing these Solidity smart contracts. This particular Solidity smart contract also implements an `update` method, to handle the contract's behaviour over time. This Solidity smart contract also requires keeping track of `or` combinator choice's in a similar manner to the SmartFin smart contract implementation; this suggests that the more complicated the SmartFin contract becomes, the closer the Solidity smart contract will be to the SmartFin smart contract implementation as there will be no superfluous data/methods due to the use of every combinator.

The issues with implementing this Solidity smart contract are generally similar to the issues with the simple smart contract, as most crop up while implementing the boilerplate. One extra issue is the fact that time is dealt with, which can be a tripping point as Ethereum smart contracts

can only access *block-time* - i.e. the time the latest block was mined at - not the current time. This issue also exists in the SmartFin smart contract, however, so it does not present a difference between the two.

Accuracy

The accuracy with which the Solidity smart contract represents the *European option* contract is also quite similar to the SmartFin smart contract. Both contracts require the holder to acquire the contract, set their `or` combinator choice, and update, although the Solidity smart contract will throw an error and revert if the `or` choice is not set whereas the SmartFin smart contract will simply do nothing. As such, there is not much improvement to writing a *European option* contract in Solidity compared to creating a SmartFin smart contract with regards to accuracy.

Efficiency and Gas Costs

The gas costs of executing certain methods on the *European option* smart contract instances were measured and recorded in table 7.2.

Gas Cost for Method Calls in Wei - <i>European option</i> Contract			
Method	SmartFin S.C.	Solidity S.C.	SmartFin to Solidity Ratio (3 s.f.)
<code>constructor</code>	15010171	2334158	6.43
<code>acquire</code>	464506	63684	7.29
<code>update</code>	294993	23712	12.4

Table 7.2: The gas costs of certain method calls on the *European option* contract’s SmartFin and Solidity smart contract instances.

Compared to the *simple contract*, the differences in gas costs between method calls on the two smart contract implementations of the *European option* contract are much more pronounced. The `constructor` call for the SmartFin smart contract was 6.43 times more expensive in terms of gas cost compared to the Solidity smart contract, likely due to the serialization and deserialization required for all of the combinators in the SmartFin smart contract.

The `acquire` method’s gas cost for the SmartFin smart contract was also 7.29 times higher than that of the Solidity smart contract. This is likely due to the recursive nature of the SmartFin smart contract’s combinator representation, requiring all combinator objects to have their `acquire` method called and have an acquisition time set. In contrast, the Solidity smart contract only needed to set a single acquired boolean thanks to the relatively simple acquisition process for the *European option* contract, which would be much cheaper. The `update` method’s higher gas cost is likely caused by similar factors, as the `update` call would require a recursive call to be initiated for the combinators, whereas the call to update in the Solidity smart contract only needs to check/modify a few variables.

Overall, for a contract with more combinators it is clear that a bespoke Solidity smart contract implementation can be significantly more efficient than a SmartFin smart contract in terms of gas costs.

7.3.3 Loan with Variable Repayment

The *loan with variable repayment* example contract is defined as follows:

```
truncate <01/01/2020 00:00:00> and
  one
  anytime then
    truncate <01/02/2020 00:00:00> give scale 2 one
    truncate <01/03/2020 00:00:00> give scale 3 one
```

This contract represents a loan from the holder to the counter-party of 1 Wei, which can be paid back with 2 Wei by February 2020, or 3 Wei by March 2020. This enables the comparison of the rest of the SmartFin combinators (including time based combinators like **then**) in a financial smart contract against the bespoke implementation of a Solidity equivalent. The Solidity implementation of the *loan with variable repayment* is listed in appendix [D](#).

Ease of Implementation

Implementing the *loan with variable repayment* contract as a SmartFin contract is relatively easy. The loan has an expiration date, requiring the **truncate** combinator. It also gives the holder 1 Wei, and then requires a repayment - these two sub-contracts can be represented by an **and** combinator. The repayment subcontract requires repayment at some point in time that the holder can decide, with different values depending on the time. This can be represented easily by an **anytime** combinator followed by a **then** combinator, with two payments from the holder (**give**) with different values (**scale x one**) and different deadlines (**truncate**). Overall, this contract can be represented in 13 combinators.

The Solidity smart contract implementation of the *loan with variable repayment* contract again repeats a large amount of boilerplate from the previous two Solidity smart contracts, and requires 207 lines to implement in total. The three relevant horizons are kept track of, as well as the **anytime** sub-contract's acquisition time. Again, this suggests that when more combinators are used in a contract, the SmartFin smart contract and Solidity smart contract representations may become closer in implementation as they will be storing similar required state. The Solidity smart contract for the *loan with variable repayment* contract does not have any unique implementation issues that the other two contracts did not encounter, besides the subtleties of dealing with the **anytime** combinator - to ensure that if its horizon is passed it is treated as if it was acquired at the horizon.

Accuracy

Again, the operation of the two smart contract representations of the *loan with variable repayment* contract by an external user is effectively the same - where the contract must be deployed, acquired, have the **anytime** sub-combinator be acquired, and be updated - and so the two representations have similar levels of accuracy.

Efficiency and Gas Costs

The gas costs of executing certain methods on the *loan with variable repayment* smart contract instances were measured and recorded in table [7.3](#).

The ratio of the gas cost of the constructor call for the SmartFin smart contract for that of the Solidity smart contract is similar to that of the *European option* contract (6.74 vs 6.43), likely because the SmartFin smart contracts for both contracts store a similar amount of state for the combinators, as do the two Solidity smart contracts - so the reasoning here is similar to that of the previous contract. The acquisition gas cost ratio was slightly higher here, possible due to the storage operations required for acquisition of the **anytime** combinator. The **anytime** sub-contract's acquisition gas cost 7.06 times higher for the SmartFin smart contract than the Solidity smart contract. Both of these methods call **update** at their completion, and the recursive nature of

Gas Cost for Method Calls in Wei - <i>loan with variable repayment</i>			
Method	SmartFin S.C.	Solidity S.C.	SmartFin to Solidity Ratio (3 s.f.)
<code>constructor</code>	16529563	2452810	6.74
<code>acquire</code>	1439031	149220	9.64
<code>acquire_anytime_-sub_combinator</code>	955180	135204	7.06

Table 7.3: The gas costs of certain method calls on the *loan with variable repayment* contract’s SmartFin and Solidity smart contract instances.

the `update` method on the SmartFin smart contract will result in lower efficiency than the simple method on the Solidity smart contract. Yet again, this contract shows that the generic nature of the SmartFin smart contracts results in lower efficiency than bespoke Solidity smart contract implementations.

7.3.4 Case Study Conclusions

Overall, it is clear that each of these contracts is significantly easier to implement using the web client as opposed to writing a Solidity smart contract. The ability to define the contract only in SmartFin and then obtain a smart contract implementation requires the user to only write a few combinators in order to obtain a SmartFin smart contract, whereas bespoke Solidity equivalents require writing 100-200 lines for these small-to-medium sized contracts. The risk of introducing errors through reentrancy or overflow/underflow bugs is also high when implementing in Solidity, whereas the pre-existing tests for the SmartFin smart contract implementation provide some confidence that less errors will be present.

The accuracy of representation for SmartFin smart contracts is very similar to that of the Solidity smart contracts; if treating the two as black boxes, a user would interact with either black box in effectively the same way. This suggests that the compromises made for the SmartFin smart contract implementation are unavoidable on any Ethereum smart contract representation of a SmartFin contract definition.

The gas costs of bespoke Solidity smart contracts are significantly lower than those of their SmartFin smart contract equivalents. This is due to the generic and recursive nature of the SmartFin smart contract implementation, which requires superfluous state and methods to be stored, and extra processing to occur on the smart contract. In context, however, none of the gas values seen ever approach a noticeable cost for the user - the highest gas cost seen being 16529563 Wei for deploying the *loan with variable repayment* SmartFin smart contract. 0.03 USD is worth approximately 10^{14} Wei at the time of writing, for context[5]. Furthermore, these benefits may be lost when implementing more complicated SmartFin contract definitions, as both smart contracts will need to store a lot more state, and the SmartFin smart contracts’ state will hold less superfluous information.

7.3.5 Case Study Remarks

When comparing SmartFin smart contracts to bespoke Solidity smart contracts, the implementation of SmartFin smart contracts was found to be significantly simpler than the implementation of equivalent Solidity smart contracts, and the risk of introducing errors in the Solidity implementation was higher due to the freedom the implementer has access to. Both implementations had a similar level of accuracy compared to the underlying financial contract, but the gas costs for SmartFin smart contracts were significantly higher than equivalent Solidity smart contracts. This is not always relevant, however, as financial institutions are more likely to use private blockchains which don’t distribute gas fees based on code execution. Furthermore, the gas costs measured were still on a minute scale in comparison to \$1 USD, and as such the cost of executing these SmartFin smart contracts is still extremely low.

7.4 Remarks

Overall, the SmartFin smart contract implementation and web client are effective tools for easily deploying smart contract representations of financial contracts. These financial smart contracts implement sufficient functionality to represent financial contracts, with minimal compromises made. The use of almost 375 automated tests helps to ensure the correctness of the SmartFin smart contract implementation. The design of SmartFin allows similar financial contract functionality defined in the original DSL by Peyton Jones et al.[\[27\]](#), with some changes to allow compatibility with the Ethereum platform. Compared to other existing options for creating financial smart contracts, like writing bespoke Solidity implementations, the ease of implementation and lower risk of erroneous behaviour make SmartFin smart contracts a more attractive option. In cases where efficiency and gas costs are paramount, however, then bespoke financial smart contracts will always have the upper hand over generic financial smart contracts - although the cost of implementing these bespoke smart contracts may not be worth the efficiency gains.

Chapter 8

Conclusions

In this work we have presented a system for defining financial contracts in the domain-specific language SmartFin, evaluating them, deploying their corresponding smart contract representations to the blockchain, and interacting with these deployed financial smart contracts. The smart contract representations of SmartFin financial smart contracts is sufficiently accurate with few compromises, and all major functionality is thoroughly tested to ensure correctness.

The implementation of these tools show that many financial contracts *can* be represented with sufficient accuracy by a smart contract implementation, despite discrepancies caused by the nature of smart contracts, and that this does not necessarily require significant work by the financial contract author. The generic smart contract implementation of SmartFin can instead allow financial smart contract authors to focus only on the implementation of the SmartFin financial contract, and to ignore the smart contract implementation. This method of development can be significantly easier on the financial smart contract author. Robustness of these financial smart contracts is also improved, as the generic smart contract implementation can be thoroughly tested with no extra effort, meaning that individual financial smart contracts do not need to undergo testing as long as the SmartFin contract definition is fully tested.

SmartFin financial contracts can also be evaluated relatively easily, compared to traditional financial smart contracts, thanks to the simple syntax of the DSL. This enables financial contracts to be evaluated before publishing, thus allowing financial contract authors to check their contracts for erroneous behaviour. Any implementation of an evaluation algorithm for SmartFin will apply to any SmartFin financial contract, greatly reducing the work required to evaluate *any* specific contract; traditional financial contracts and bespoke financial smart contracts would instead need to be evaluated individually, through complex logical and mathematical analysis that is difficult to automate.

The SmartFin DSL’s functional and combinatorial nature can be represented quite elegantly using a simple tree of trait objects; this maintains the compositional and encapsulated nature of each combinator, while still keeping track of all state required during program execution.

While SmartFin may not fit the needs of every financial institution, the findings presented can be generalised to apply to any similar combinatorial financial contract DSL. Implementing a generic smart contract that can represent any financial contract written in a specific DSL has all of the benefits stated here for SmartFin, including ease of implementation, improved robustness, and ease of evaluation. While bespoke smart contracts may be more efficient in terms of gas costs, the development efforts saved by using a financial contract DSL implementation can greatly outweigh the efficiency gains.

We hope that these findings can be applied in the world of financial engineering, allowing financial institutions to implement financial smart contracts with significantly less effort than a bespoke

approach on many fronts.

8.1 Reflections

When it comes to developing the smart contract which implements SmartFin, there are definitely different approaches that could have been taken. Some of the issues discussed regarding the implemented smart contract in chapter 7, like the requirement of function arguments to be serialized and the inefficient usage of storage, came about mainly as a result of using Rust and the `pwasm` modules. There were also some other issues with this approach which impacted development, including difficulty with building the smart contract on different systems from the same repository, and difficulty debugging due to bugs in the `pwasm` modules' implementations. The `pwasm` modules are still in active development and have not yet had a 1.0 release, so some of this is to be expected.

While these issues are inconvenient to say the least, different approaches to implementation are not perfect either; implementing the SmartFin smart contract in Solidity, for instance, would require much more care to avoid vulnerabilities like integer overflow/underflow on arithmetic operations, and would require a more complicated development setup automated testing. As such, there is no perfect approach to implementing this smart contract, but it may have been beneficial to perform some hands-on evaluation of different approaches rather than relying mainly on research; this could have allowed a more ideal development process to have been used, resulting in a more ideal final product.

After developing in the blockchain ecosystem for several months, certain insights can be gained as to issues with the field and some of their potential solutions. It is clear that blockchain development is still cutting-edge, but not necessarily for all the right reasons. One of the aspects of blockchain development which caused a significant amount of friction during development is the ecosystem of tools and technologies; while there are many tools to be used, most are incomplete or in early development, and many tools tend to only be usable in very specific circumstances. The specificity of tools seems to be a result of the relatively fragmented nature of blockchain development as a field, where what works on Ethereum blockchains won't always work on Bitcoin or Ripple blockchains. Blockchain development tools would be much more useful if they could be used with different types of blockchains, which could be possible if blockchains followed some set of standards - similar to how browsers follow web standards devised by the W3C[48].

Besides their specificity, tools for blockchain development also tend to have lacking, outdated, or extremely technical documentation - where a user will struggle or be completely unable to understand how to use a given tool without prior knowledge. This may be partially because blockchain as a platform is still in its relatively early days, but improved documentation would go a long way to opening up blockchain development to many more developers.

8.2 Areas for Future Work

While the essential functionality of SmartFin's smart contract implementation is implemented in this project, there are a number of areas that could be further developed. The findings in these areas could be useful both for users of SmartFin and for financial institutions as a whole.

8.2.1 Maximal SmartFin Contract Evaluation and Numerical Modelling

Currently the web client allows the user to perform step-by-step evaluation of a SmartFin contract definition, setting the times at which the contract and sub-contracts are acquired, and the branches taken for `or` combinator sub-contracts (see section 6.2.8 for more details). While this is useful for testing financial contracts, it would be even more useful to provide some maximal/minimal evaluation functionality. This could show the author of a financial contract how much the contract could possible gain or cost, highlighting any egregious errors which could lose one party a significant

amount of funds.

The web client’s evaluation framework (the `Evaluator` class described in section 6.2.8) could be used for maximal and minimal evaluation with some extra implementation effort, but the biggest problem with this is the value of observables. How can an algorithm decide whether one observable’s value is bigger than another, especially when these values may not be known in the real world yet?

To handle this issue, the web client could provide some numerical modelling functionality. The user could provide the details required to form a numerical model of the value of each observable over time, and the web client could base comparisons between observables on the average expected value of the observables in the given time period.

There are several methods of implementing this numerical modelling. One option would be a lattice-based model, which uses a binomial tree to represent the probabilistic values a measurement can take over time[28]. In the paper by Peyton Jones et al.[27] where the original combinator DSL is defined, lattice-based modelling is used to evaluate the financial contract definition with regards to all probabilistic influences, including interest rates, exchange rates, and the value of observables; as such, a similar implementation of this is shown to be possible (albeit very complicated).

8.2.2 Expansion of the SmartFin DSL

While the SmartFin DSL does allow users to represent a large portion of financial contracts, not *every* financial contract can be represented this way[27]. For a financial institution, the benefits to using the SmartFin DSL to compose financial contracts will always apply, but financial contracts which must instead be represented by complicated prose with a bespoke Solidity smart contract implementation will not stand to gain from this work. These bespoke smart contracts must be tested and analysed individually to prevent easy-to-miss errors from effecting the contract’s behaviour, and their implementation can require hundreds of lines in comparison to tens of combinators.

Expanding the SmartFin DSL and its smart contract implementation serves to reduce the number of bespoke financial smart contracts and verbose traditional financial contracts that need to be implemented. In doing this, financial institutions can leverage SmartFin to produce financial contracts with less development effort, or can learn from the combinators employed to design their own DSL for a similar purpose.

In terms of the directions in which SmartFin should be expanded, it is quite difficult to design new combinators. After a discussion with Dr. Panos Parpas of Imperial College London, a former professional financial contract author, one area of financial contracts which was notably absent from the original DSL presented by Peyton Jones et al.[27] is comparison; there are no combinators in the DSL which specifically handle comparisons between values. While this could be encoded in the definition of an observable, this usage of observables retreats back into the world of verbose and complex financial contracts, making evaluation and smart contract representation more difficult. As such, specific encoding of comparison operators in SmartFin would maintain the elegance of SmartFin financial contracts while expanding upon functionality.

Appendix A

SmartFin Smart Contract ABI

```
1 // The financial smart contract interface
2 #[eth_abi(FinancialScEndpoint)]
3 pub trait FinancialScInterface {
4     // The contract constructor, takes the combinator contract definition (
5     //     ↳ serialized) and the holder address
6     fn constructor(&mut self, contract_definition: Vec<i64>, holder: Address,
7     //     ↳ use_gas: bool);
8
9     // Gets the address of the contract holder
10    #[constant]
11    fn get_holder(&mut self) -> Address;
12
13    // Gets the address of the counter-party
14    #[constant]
15    fn get_counter_party(&mut self) -> Address;
16
17    // Gets the combinator contract definition, returns the combinator contract
18    //     ↳ serialized
19    #[constant]
20    fn get_contract_definition(&mut self) -> Vec<i64>;
21
22    // Gets the current balance of the given party (true is holder, false counter-
23    //     ↳ party)
24    #[constant]
25    fn get_balance(&mut self, holderBalance: bool) -> i64;
26
27    // Gets whether or not the contract has concluded all operation (i.e. updating
28    //     ↳ will never change the balance).
29    #[constant]
30    fn get_concluded(&mut self) -> bool;
31
32    // Gets whether or not the contract allocates gas fees upon withdrawal.
33    #[constant]
34    fn get_use_gas(&mut self) -> bool;
35
36    // Gets the last-updated time.
37    #[constant]
38    fn get_last_updated(&mut self) -> i64;
39
40    // Gets the contract acquisition times (top level acquisition time and anytime
41    //     ↳ acquisition times)
42    #[constant]
43    fn get_acquisition_times(&mut self) -> Vec<i64>;
44
45    // Gets the or choices
46    #[constant]
47    fn get_or_choices(&mut self) -> Vec<u8>;
48
49    // Gets the concrete observable values
50    #[constant]
51    fn get_obs_entries(&mut self) -> Vec<i64>;
52
53    // Sets the preference of the given or combinator's sub-combinators
```

```

48     fn set_or_choice(&mut self, or_index: u64, choice: bool);
49
50     // Sets a value for the given observable
51     fn set_obs_value(&mut self, obs_index: u64, value: i64);
52
53     // Acquires the combinator contract at the current block-time (when called by
54     //   ↳ the holder)
55     fn acquire(&mut self);
56
57     // Updates the balances of the holder and counter-party
58     fn update(&mut self);
59
60     // Acquires an anytime combinator's sub-contract
61     fn acquire_anytime_sub_contract(&mut self, anytime_index: u64);
62
63     // Stakes Eth with the contract (can be called by the holder or counter-party),
64     //   ↳ returns the caller's total balance
65     #[payable]
66     fn stake(&mut self) -> i64;
67
68     // Withdraws positive Eth balance up to the given amount from the contract (can
69     //   ↳ be called by the holder or counter-party)
70     fn withdraw(&mut self, amount: u64) ;
71 }

```

Listing A.1: The ABI of the smart contract implementation of SmartFin, defined as a trait in Rust¹. See section 4.2.2 for details.

¹Rust syntax highlighting derived from the `solidity-latex-highlighting` package package written by Sergei Tikhomirov, used under the MIT license, available at <https://github.com/s-tikhomirov/solidity-latex-highlighting>.

Appendix B

Solidity Implementation of the *One* Case Study

```
1 pragma solidity >=0.4.22 <0.6.0;
2
3 // Represents the contract 'one'
4 contract One {
5     // Static values
6     int256 MAX_INT256 = int256(~(uint256(1) << 255));
7     int256 MIN_INT256 = int256(uint256(1) << 255);
8
9     // The contract holder
10    address holder;
11
12    // The counter-party
13    address counterParty;
14
15    // The stakes of the holder and counter-party
16    mapping(address => int256) stakes;
17
18    // Whether or not this contract has been acquired
19    bool acquired;
20
21    // Constructor, takes the contract holder address
22    constructor(address contractHolder) public {
23        require(
24            contractHolder != msg.sender,
25            "Holder and counter-party cannot have the same address."
26        );
27        // Set the holder and counter-party
28        holder = contractHolder;
29        counterParty = msg.sender;
30
31        // Initialise stakes to 0 and acquired to false
32        stakes[counterParty] = 0;
33        stakes[holder] = 0;
34        acquired = false;
35    }
36
37    // Only allows the holder or counter-party to call a function
38    modifier onlyParties() {
39        require(
40            msg.sender == counterParty || msg.sender == holder,
41            "This function can only be called by the holder or the counter-party."
42        );
43
44        -;
45    }
46
47    // Returns the balance of one of the two parties
48    function getBalance(bool holderBalance) public view returns (int256) {
49        if (holderBalance) {
50            return stakes[holder];
```

```

51     } else {
52         return stakes[counterParty];
53     }
54 }
55
56 // Acquires this contract
57 function acquire() public {
58     require(
59         msg.sender == holder,
60         "Only the holder may call this function."
61     );
62     require(
63         !acquired,
64         "This function can only be called before acquisition."
65     );
66
67     acquired = true;
68
69     // Update balances
70     transferToHolder(1);
71 }
72
73 // Stake Ether in the contract
74 function stake() public payable onlyParties() {
75     require(
76         uint256(MAX_INT256) >= msg.value,
77         "Value being staked is too big to be stored as an int256 value."
78     );
79
80     // Update balance
81     stakes[msg.sender] = safeAddSigned(stakes[msg.sender], int256(msg.value));
82 }
83
84 // Withdraw Ether from the contract
85 function withdraw(uint64 amount) public onlyParties() {
86     require(
87         address(this).balance > 0,
88         "Contract does not have enough funds."
89     );
90     require(
91         stakes[msg.sender] > 0,
92         "The caller does not have enough stake."
93     );
94
95     uint64 finalAmount = amount;
96
97     // Clamp withdrawal amount to total contract balance
98     if (address(this).balance < finalAmount) {
99         finalAmount = uint64(address(this).balance);
100     }
101
102     // Clamp withdrawal amount to party's balance
103     if (stakes[msg.sender] < finalAmount) {
104         finalAmount = uint64(stakes[msg.sender]);
105     }
106
107     // Adjust balance first to prevent re-entrancy bugs
108     stakes[msg.sender] = safeSubSigned(stakes[msg.sender], int256(finalAmount))
109     ↪ ;
110
111     // Send Ether (with no gas)
112     msg.sender.call.value(finalAmount).gas(0);
113 }
114
115 // Transfers the given amount from the holder to the counter-party
116 function transferToHolder(int256 amount) private {
117     stakes[holder] = safeAddSigned(stakes[holder], amount);
118     stakes[counterParty] = safeSubSigned(stakes[counterParty], amount);
119 }
120
121 // Add two signed integers if no overflow or underflow can occur
122 function safeAddSigned(int256 a, int256 b) private view returns (int256) {
123     require(

```

```

123         (b >= 0 && a <= MAX_INT256 - b) ||
124         (b < 0 && a >= MIN_INT256 - b),
125         "Integer overflow or underflow."
126     );
127
128     return a + b;
129 }
130
131 // Subtract one signed integer from another if no overflow or underflow can
132 //    ↪ occur
133 function safeSubSigned(int256 a, int256 b) private view returns (int256) {
134     require(
135         b != MIN_INT256,
136         "Integer overflow or underflow."
137     );
138     return safeAddSigned(a, -b);
139 }
140 }

```

Listing B.1: A Solidity smart contract for the *one* contract case study in section 7.3.

Appendix C

Solidity Implementation of the *European Option* Case Study

```
1 pragma solidity >=0.4.22 <0.6.0;
2
3 // Represents the contract 'get truncate <01/01/2020 00:00:00> or one zero'
4 contract EuroOption {
5     // A struct representing an or-choice
6     struct OrChoice {
7         // Whether or not the or-choice has been set
8         bool set;
9
10        // The or-choice (true is the first sub-contract, false is the second sub-
11        // ↪ contract)
12        bool choice;
13    }
14
15    // Static values
16    int256 MAX_INT256 = int256(~(uint256(1) << 255));
17    int256 MIN_INT256 = int256(uint256(1) << 255);
18    uint256 HORIZON_UNIX = 1577836800;
19
20    // The contract holder
21    address holder;
22
23    // The counter-party
24    address counterParty;
25
26    // The stakes of the holder and counter-party
27    mapping(address => int256) stakes;
28
29    // Whether or not this contract has been acquired
30    bool acquired = false;
31
32    // Whether or not this contract has been fully-updated
33    bool fullyUpdated = false;
34
35    // The or-choice
36    OrChoice orChoice = OrChoice(false, false);
37
38    // Constructor, takes the contract holder address
39    constructor(address contractHolder) public {
40        require(
41            contractHolder != msg.sender,
42            "Holder and counter-party cannot have the same address."
43        );
44        // Set the holder and counter-party
45        holder = contractHolder;
46        counterParty = msg.sender;
47
48        // Initialise stakes to 0
49        stakes[counterParty] = 0;
50        stakes[holder] = 0;
```

```

50 }
51
52 // Only allows the holder or counter-party to call a function
53 modifier onlyParties() {
54     require(
55         msg.sender == counterParty || msg.sender == holder,
56         "This function can only be called by the holder or the counter-party."
57     );
58
59     -;
60 }
61
62 // Only allows the holder to call a function
63 modifier onlyHolder() {
64     require(
65         msg.sender == holder,
66         "Only the holder may call this function."
67     );
68
69     -;
70 }
71
72 // Returns the balance of one of the two parties
73 function getBalance(bool holderBalance) public view returns (int256) {
74     if (holderBalance) {
75         return stakes[holder];
76     } else {
77         return stakes[counterParty];
78     }
79 }
80
81 // Sets the choice for the or-combinator
82 function setOrChoice(bool firstSubContract) public onlyHolder() {
83     require(
84         !orChoice.set,
85         "The or-choice has already been set."
86     );
87
88     orChoice.set = true;
89     orChoice.choice = firstSubContract;
90 }
91
92 // Acquires this contract
93 function acquire() public onlyHolder() {
94     require(
95         !acquired,
96         "This function can only be called before acquisition."
97     );
98     require(
99         now <= HORIZON_UNIX,
100         "This contract can only be acquired until 01/01/2020 00:00:00 UTC."
101     );
102
103     acquired = true;
104 }
105
106 // Updates the contract balance
107 function update() public {
108     require(
109         acquired,
110         "The contract must be acquired before updating."
111     );
112     require(
113         !fullyUpdated,
114         "The contract must not be fully-updated when updating."
115     );
116     require(
117         orChoice.set,
118         "The or-choice must be set before updating."
119     );
120
121     // Must be at or past the horizon for get to acquire sub-contract
122     if (now < HORIZON_UNIX) {

```



```

123         return;
124     }
125
126     // If or-choice is true, acquire one
127     if (orChoice.choice) {
128         transferToHolder(1);
129     }
130 }
131
132 // Stake Ether in the contract
133 function stake() public payable onlyParties() {
134     require(
135         uint256(MAX_INT256) >= msg.value,
136         "Value being staked is too big to be stored as an int256 value."
137     );
138
139     // Update balance
140     stakes[msg.sender] = safeAddSigned(stakes[msg.sender], int256(msg.value));
141 }
142
143 // Withdraw Ether from the contract
144 function withdraw(uint64 amount) public onlyParties() {
145     require(
146         address(this).balance > 0,
147         "Contract does not have enough funds."
148     );
149     require(
150         stakes[msg.sender] > 0,
151         "The caller does not have enough stake."
152     );
153
154     uint64 finalAmount = amount;
155
156     // Clamp withdrawal amount to total contract balance
157     if (address(this).balance < finalAmount) {
158         finalAmount = uint64(address(this).balance);
159     }
160
161     // Clamp withdrawal amount to party's balance
162     if (stakes[msg.sender] < finalAmount) {
163         finalAmount = uint64(stakes[msg.sender]);
164     }
165
166     // Adjust balance first to prevent re-entrancy bugs
167     stakes[msg.sender] = safeSubSigned(stakes[msg.sender], int256(finalAmount))
168     ↪ ;
169
170     // Send Ether (with no gas)
171     msg.sender.call.value(finalAmount).gas(0);
172 }
173
174 // Transfers the given amount from the holder to the counter-party
175 function transferToHolder(int256 amount) private {
176     stakes[holder] = safeAddSigned(stakes[holder], amount);
177     stakes[counterParty] = safeSubSigned(stakes[counterParty], amount);
178 }
179
180 // Add two signed integers if no overflow or underflow can occur
181 function safeAddSigned(int256 a, int256 b) private view returns (int256) {
182     require(
183         (b >= 0 && a <= MAX_INT256 - b) ||
184         (b < 0 && a >= MIN_INT256 - b),
185         "Integer overflow or underflow."
186     );
187
188     return a + b;
189 }
190
191 // Subtract one signed integer from another if no overflow or underflow can
192 ↪ occur
193 function safeSubSigned(int256 a, int256 b) private view returns (int256) {
194     require(
195         b != MIN_INT256,

```

```
194         "Integer overflow or underflow."
195     );
196
197     return safeAddSigned(a, -b);
198 }
199 }
```

Listing C.1: A Solidity smart contract for the *European option* contract case study in section [7.3](#).

Appendix D

Solidity Implementation of the *Loan with Variable Repayment* Case Study

```
1  pragma solidity >=0.4.22 <0.6.0;
2
3  // Represents the contract:
4  // truncate <01/01/2020 00:00:00> and
5  //     one
6  //     anytime then
7  //         truncate <01/02/2020 00:00:00>
8  //             give scale 2 one
9  //         truncate <01/03/2020 00:00:00>
10 //             give scale 3 one
11 contract LoanVarRepay {
12     // A struct representing an anytime sub-contract acquisition time
13     struct AnytimeAcquisitionTime {
14         // Whether or not the sub-contract has been acquired
15         bool acquired;
16
17         // The acquisition time
18         uint256 time;
19     }
20
21     // Static values
22     int256 MAX_INT256 = int256(~(uint256(1) << 255));
23     int256 MIN_INT256 = int256(uint256(1) << 255);
24     uint256 HORIZON_UNIX_1 = 1577836800;
25     uint256 HORIZON_UNIX_2 = 1580515200;
26     uint256 HORIZON_UNIX_3 = 1583020800;
27
28     // The contract holder
29     address holder;
30
31     // The counter-party
32     address counterParty;
33
34     // The stakes of the holder and counter-party
35     mapping(address => int256) stakes;
36
37     // Whether or not this contract has been acquired
38     bool acquired = false;
39
40     // Whether or not this contract has been fully-updated
41     bool fullyUpdated = false;
42
43     // The anytime sub-contract acquisition time
44     AnytimeAcquisitionTime subAcquisitionTime = AnytimeAcquisitionTime(false, 0);
45
46     // Constructor, takes the contract holder address
47     constructor(address contractHolder) public {
```

```

48     require(
49         contractHolder != msg.sender,
50         "Holder and counter-party cannot have the same address."
51     );
52     // Set the holder and counter-party
53     holder = contractHolder;
54     counterParty = msg.sender;
55
56     // Initialise stakes to 0
57     stakes[counterParty] = 0;
58     stakes[holder] = 0;
59 }
60
61 // Only allows the holder or counter-party to call a function
62 modifier onlyParties() {
63     require(
64         msg.sender == counterParty || msg.sender == holder,
65         "This function can only be called by the holder or the counter-party."
66     );
67
68     -;
69 }
70
71 // Only allows the holder to call a function
72 modifier onlyHolder() {
73     require(
74         msg.sender == holder,
75         "Only the holder may call this function."
76     );
77
78     -;
79 }
80
81 // Returns the balance of one of the two parties
82 function getBalance(bool holderBalance) public view returns (int256) {
83     if (holderBalance) {
84         return stakes[holder];
85     } else {
86         return stakes[counterParty];
87     }
88 }
89
90 // Acquires the anytime sub-contract
91 function acquireAnytimeSubContract() public onlyHolder() {
92     require(
93         !subAcquisitionTime.acquired,
94         "The anytime sub-contract has already been acquired."
95     );
96
97     subAcquisitionTime.acquired = true;
98     subAcquisitionTime.time = now;
99     this.update();
100 }
101
102 // Acquires this contract
103 function acquire() public onlyHolder() {
104     require(
105         !acquired,
106         "This function can only be called before acquisition."
107     );
108     require(
109         now <= HORIZON_UNIX_1,
110         "This contract can only be acquired until 01/01/2020 00:00:00 UTC."
111     );
112
113     acquired = true;
114     transferToHolder(1);
115 }
116
117 // Updates the contract balance
118 function update() public {
119     require(
120         acquired,

```

```

121         "The contract must be acquired before updating."
122     );
123     require(
124         !fullyUpdated,
125         "The contract must not be fully-updated when updating."
126     );
127     require(
128         subAcquisitionTime.acquired || now > HORIZON_UNIX_3,
129         "The anytime sub-contract must be acquired before updating."
130     );
131
132     uint256 repayTime = (subAcquisitionTime.acquired) ? subAcquisitionTime.time
133         ↪ : HORIZON_UNIX_3;
134
135     if (repayTime <= HORIZON_UNIX_2) {
136         transferToHolder(-2);
137     } else {
138         transferToHolder(-3);
139     }
140
141     // Stake Ether in the contract
142     function stake() public payable onlyParties() {
143         require(
144             uint256(MAX_INT256) >= msg.value,
145             "Value being staked is too big to be stored as an int256 value."
146         );
147
148         // Update balance
149         stakes[msg.sender] = safeAddSigned(stakes[msg.sender], int256(msg.value));
150     }
151
152     // Withdraw Ether from the contract
153     function withdraw(uint64 amount) public onlyParties() {
154         require(
155             address(this).balance > 0,
156             "Contract does not have enough funds."
157         );
158         require(
159             stakes[msg.sender] > 0,
160             "The caller does not have enough stake."
161         );
162
163         uint64 finalAmount = amount;
164
165         // Clamp withdrawal amount to total contract balance
166         if (address(this).balance < finalAmount) {
167             finalAmount = uint64(address(this).balance);
168         }
169
170         // Clamp withdrawal amount to party's balance
171         if (stakes[msg.sender] < finalAmount) {
172             finalAmount = uint64(stakes[msg.sender]);
173         }
174
175         // Adjust balance first to prevent re-entrancy bugs
176         stakes[msg.sender] = safeSubSigned(stakes[msg.sender], int256(finalAmount))
177             ↪ ;
178
179         // Send Ether (with no gas)
180         msg.sender.call.value(finalAmount).gas(0);
181     }
182
183     // Transfers the given amount from the holder to the counter-party
184     function transferToHolder(int256 amount) private {
185         stakes[holder] = safeAddSigned(stakes[holder], amount);
186         stakes[counterParty] = safeSubSigned(stakes[counterParty], amount);
187     }
188
189     // Add two signed integers if no overflow or underflow can occur
190     function safeAddSigned(int256 a, int256 b) private view returns (int256) {
191         require(
192             (b >= 0 && a <= MAX_INT256 - b) ||

```

```

192         (b < 0 && a >= MIN_INT256 - b),
193         "Integer overflow or underflow."
194     );
195
196     return a + b;
197 }
198
199 // Subtract one signed integer from another if no overflow or underflow can
200 //    ↪ occur
201 function safeSubSigned(int256 a, int256 b) private view returns (int256) {
202     require(
203         b != MIN_INT256,
204         "Integer overflow or underflow."
205     );
206     return safeAddSigned(a, -b);
207 }
208 }

```

Listing D.1: A Solidity smart contract for the *loan with variable repayment* contract case study in section 7.3.

Appendix E

User Manual

E.1 Getting Started

E.1.1 Overview

The financial smart contract client is a client which allows the user to define a financial contract in a domain-specific language called SmartFin, and publish it to a blockchain in the form of a smart contract. These financial contracts can be as simple as one party paying the other 1 Wei (the minimal amount of currency payable on an Ethereum blockchain), but they can also be more complex - like a European option. For more information on writing financial contracts in SmartFin, see section [E.2](#).

The client can be connected to a blockchain of your choice (see sections [E.1.2](#) and [E.3](#)), and used to publish *financial smart contracts* (smart contracts which represent SmartFin financial contracts). These contracts are made between two parties. One party is the *holder* - the party that can sign, or *acquire*, the contract. The second party is the *counter-party* - the party that authors the contract. It is not currently possible to create a contract with any number of parties other than 2 in this client.

When the holder *acquires* a financial smart contract, the contract will track the balance of Wei between the two parties. For example, a smart contract representing a SmartFin financial contract requiring the counter-party to pay the holder 1 Wei immediately will track the holder's balance increasing by 1 Wei, and the counter-party's balance decreasing by 1 Wei, upon acquisition by the holder.

This manual contains instructions on using the financial smart contract client to compose and evaluate SmartFin financial contracts, and deploy and interact with financial smart contracts.

E.1.2 Installation

In order to use the financial smart contract client, run the provided `fsc-server.sh` file (requires Python 3[\[23\]](#) and Google Chrome[\[25\]](#)). This will run a local server, and open a tab in Google Chrome to the client page.

If no tab is opened automatically, open a browser window in Google Chrome and navigate to `http://localhost:8080`.

E.2 SmartFin - The Financial Contract DSL

E.2.1 Overview

The financial smart contract client deals with smart contracts that represent financial contracts defined in SmartFin, a combinator language used to describe financial contracts. A combinator language is a functional language in which a program is made up of chained function calls. As such, each contract written in SmartFin can be used as a sub-contract for any combinator. This means that a whole financial contract can be represented by a single combinator, or by some composition of combinators.

Each SmartFin financial contract has a *holder*, and a *counter-party*. Typically, the counter-party will be the party making payments, and the holder will be the party receiving payments. All payments are made in Ether (and all amounts of currency will be in the form of Wei, the smallest denomination of Ether).

A SmartFin financial contract can be *acquired* by the holder at any point in time, but the responsibilities of each party may differ depending on when the contract is acquired. For example, consider a contract C_1 which requires the counter-party to pay the holder 100 Ether on noon of January 1st 2019 and again on noon of January 1st 2020. C_1 requires 2 payments to occur if acquired before 12:00 on 01/01/19, 1 payment to occur if acquired by the 12:00 01/01/20, or none otherwise. The acquisition date of a SmartFin financial contract will therefore affect the value of the contract for each party.

A SmartFin financial contract may also *expire* where no responsibilities outlined in the contract take effect if the contract is acquired after a certain time. For example, the contract C_1 has no effect if acquired after 12:00 on 01/01/20. This date is called the *horizon* of the SmartFin contract. An important thing to note is that a contract's responsibilities could potentially extend past the contract's horizon, but a contract acquired after its horizon will have no effect.

Some SmartFin financial contracts may be dependent on not just sub-contracts, but also parameters. The contract C_1 , for instance, defines payments of a specific amount on two specific dates. This contract would need to be defined with a constant representing 100 Ether, and two date/times. A SmartFin financial contract could also be dependent on a variable value, such as the temperature in London in Celsius, or the distance between two people in metres. Such a value is called an *observable*.

E.2.2 Combinators

The set of combinators defined in SmartFin is described below, along with the type signature of each combinator (using the function signature notation of Haskell). The notation used to describe SmartFin is defined in table E.1.

c, d	<i>Contract</i>
o	<i>Observable</i>
t	<i>Date/Time</i>

Table E.1: Conventions for SmartFin's Description

`zero :: Contract`

This combinator represents a contract with no terms. It can be acquired at any time.

`one :: Contract`

This combinator represents a contract which requires the counter-party to immediately pay the holder one Wei upon acquisition. This contract can be acquired at any time.

`give :: Contract -> Contract`

`give c` represents `c` with all responsibilities reversed (e.g. if the holder acquires `give one`, they must pay the counter-party 1 Wei immediately).

`and :: Contract -> Contract -> Contract`

When `and c d` is acquired, both `c` and `d` are acquired immediately. Expired sub-contracts are not acquired.

`or :: Contract -> Contract -> Contract`

When `or c d` is acquired, the holder immediately acquires either `c` or `d`. If one has expired, the holder cannot acquire it (and must acquire the other if possible).

`truncate :: Date -> Contract -> Contract`

When `truncate t c` is acquired, the holder acquires `c`. The horizon of `truncate t c` is the earliest of `t` and the horizon of `c` (thus `truncate t c` does nothing after either horizon has passed).

Dates in SmartFin must be provided in either the format `<DD/MM/YYYY HH:mm:ss>`, the format `<DD/MM/YYYY HH:mm:ss Z>`, or in UNIX Epoch time format. For more information on how to format times, see section [E.5.1](#).

`then :: Contract -> Contract -> Contract`

When acquiring `then c d`, the holder acquires `c` if `c` has not expired, or `d` if `c` has expired and `d` has not.

`scale :: Observable -> Contract -> Contract`

`scale o c` represents `c` with all payments multiplied by the value of the observable `o` at the time of acquisition.

An observable is represented by either a number (e.g. `scale 5 one` requires the counter-party to pay 5 Wei to the holder), or by a name and address if the observable has a time-varying value. The name is used to refer to the observable in the financial smart contract client, and the address is the user address of an arbiter for the observable's value that will provide its value at some point. This is written in the form `scale <name> <addr> c`, e.g. `scale tempInLondon 0xA0a4D3524dC3428884c41C05CD344f9BcB5c79f3 one`. Observable names can be in any form as long as they contain at least 1 non-mathematical character, such as a letter.

```
get :: Contract -> Contract
```

Acquiring `get c` acquires `c` at the moment in time when the horizon of `c` is reached. For example, `get truncate t one` will require the counter-party to pay the holder 1 Wei at time `t` (if acquired before it expires).

```
anytime :: Contract -> Contract
```

After `anytime c` is acquired, `c` can be acquired by the holder at any time before it expires, and must be acquired by this point.

E.2.3 Examples

Zero-Coupon Discount Bond

One example of a simple financial contract is a *zero-coupon discount bond*. This is a contract between a holder and a counter-party that requires the counter-party to pay a specified amount of currency to the holder at a certain date.

A zero-coupon discount bond which requires the counter-party to pay 100 Wei to the holder at 12:00pm on 01/01/2020 is defined in SmartFin as:

```
get truncate <01/01/2020 12:00:00> scale 100 one
```

Once the `get` combinator is acquired, its sub-contract will be acquired at the acquisition date, i.e. 12:00pm on 01/01/01. The `truncate` combinator will not yet have expired, and so its underlying contract will be acquired at this point. The acquisition of the `scale` combinator causes its underlying contract (with values multiplied by 100) to be acquired immediately, thus acquiring the `one` combinator. This results in the counter-party paying 100 Wei to the holder at 12:00pm on 01/01/2020, if acquired before this time.

European Options

A European option is another type of financial contract, which states that the holder can choose whether or not to acquire a contract on a given date.

A European option over the contract `c` at 12:00pm on 01/01/2020 is defined in SmartFin as:

```
get truncate <01/01/2020 12:00:00> or c zero
```

Similarly to the previous contract, acquiring the `get` combinator acquires the sub-contract at its horizon, i.e. 12:00pm on 01/01/2020. This acquires the non-expired `truncate` combinator, and thus the underlying `or` combinator.

At any point in time, the holder may specify which branch of the `or` combinator they would like to acquire. In the financial smart contract implementation, the contract will not proceed until a choice is made. After a choice is made, the chosen branch is evaluated based on the acquisition time of the `or` combinator (i.e. 01/01/2020 12:00pm), regardless of when the `or` choice is actually supplied.

This means that the user will select between the underlying contract `c` and `zero`, and the result will be paid out at 01/01/2020 12:00pm, or as soon as the `or` choice is provided (whichever is latest).

E.3 Connecting to a Blockchain

E.3.1 Overview

Once the server has been started and the client is open in a browser tab, you will be greeted by the blockchain connection screen, shown in figure [E.1](#).

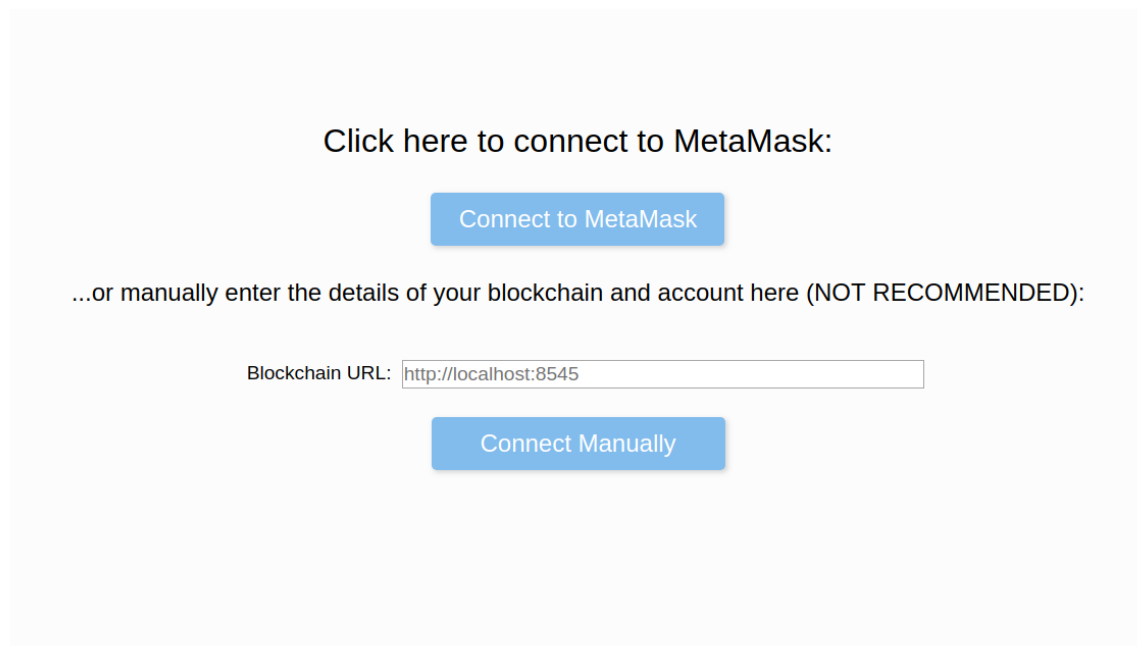


Figure E.1: The *Connect* view, through which you can connect the client to a blockchain.

If you are using MetaMask in your browser, you can connect to it using the *Connect to MetaMask* button. This will automatically detect your account's address, and allow you to approve any blockchain interaction through MetaMask.

E.3.2 Manual Connection

If you are running a local test blockchain, it may be easier to enter your blockchain's details manually. Please note, this is **not recommended** unless you are connecting to a local test blockchain, as your credentials will be processed directly through `web3` in plaintext, which is not secure.

Pressing the *Connect Manually* button will prompt the client to communicate with the blockchain at the address in the *Blockchain URL* input. If the blockchain is successfully found, you will be prompted for the account's address and password, as shown in figure [E.2](#).

If the account's details are correct, the account will be unlocked permanently (or until the connected blockchain node is restarted), and the client will proceed to the main menu.

If the connection is not successful, please ensure that the blockchain URL is correct, and that the account credentials are correct.

...or manually enter the details of your blockchain and account here (NOT RECOMMENDED):

Blockchain URL:

Address:

Password:

Figure E.2: The account detail inputs in the *Connect* view, after a manually-entered blockchain is found.

E.4 Main Menu

Once you have connected the client to a blockchain, the client will progress to the main menu (shown in figure E.3).

Figure E.3: The *Main Menu* view.

From the main menu, there are 2 available options. You may proceed to the *contract composition* view, to create, evaluate, and deploy new SmartFin financial contracts and their corresponding financial smart contracts. The other option is to proceed to the *contract monitoring* view, to monitor and interact with a deployed financial smart contract.

Press one of the two buttons to proceed to the appropriate view. You may return to the main menu from either of these menus.

E.5 Composing a SmartFin Contract

From the main menu, you may have progressed to the contract composition view (shown in figure E.4). If you progressed to the monitoring view instead, please see section E.6. To return to the main menu, press the button at the top left of the screen. The contract composition menu allows you to write, evaluate, and deploy a SmartFin financial contract and its financial smart contract counterpart.

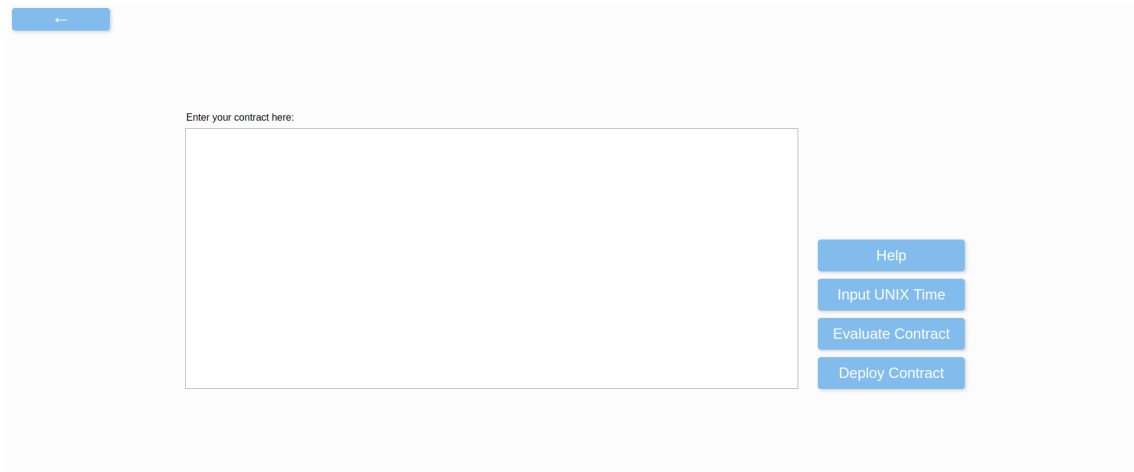


Figure E.4: The *Composition* view, consisting of a large text input on the left for a SmartFin financial contract definition, and several options to the right. The top-left button will return you to the main menu.

E.5.1 Composing a SmartFin Financial Contract

In the large text input, you may input a SmartFin financial contract. If you need to know how to write a financial contract with SmartFin, please refer to section E.2. Alternatively, the *Help* button in the monitoring view will display a similar guide for writing in SmartFin.

Inputting Time

Some SmartFin contracts require a time value to be provided, which will be a specific time on a specific date. For example, the `truncate` combinator causes a contract to expire at a given time (if not earlier). These time values can be provided in a few ways.

One way to provide times to a SmartFin contract is in the format `<DD/MM/YYYY HH:mm:ss>`. For example, the contract `truncate <01/02/2020 13:45:01> one` is worth 1 Wei up until 13:45:01 on the 1st of February 2020.

By default, the time zone for times input in this manner is set to the time zone of your client's locale. In order to change this, the time zone can be provided by inputting a time in the format `<DD/MM/YYYY HH:mm:ss Z>`, where `Z` represents a 2 or 4 digit time zone offset. For example, the time `"<01/02/2020 12:34:00 +1234>"` is equivalent to the time `"<01/02/2020 00:00:00 +0000>"`. Times returned from the client are usually displayed in UTC, with the time zone visible to prevent ambiguity.

Times can also be provided in UNIX Epoch time format. To input a UNIX Epoch time more conveniently, pressing the *Input UNIX Time* button will open a view to input the time using a calendar and time selector. Upon pressing the *Input UNIX Time* button in this menu, the UNIX Epoch time corresponding to the input date/time (in UTC) will be inserted into the SmartFin contract definition at the text cursor's current location.

Verification

Once you have input a SmartFin financial contract definition, you may verify it by pressing the *Evaluate Contract* button, or the *Deploy Contract* button (pressing **enter** is equivalent to pressing *Deploy Contract*).

Upon pressing either of these buttons (or the **enter** key), the SmartFin contract will be verified. If it contains any errors, then they will be displayed below the contract input (as shown in figure E.5). The error message can be expanded by clicking on it, which will show a more detailed trace of the error.

The screenshot shows a web interface for entering a SmartFin contract. At the top, there is a text input field with the placeholder "Enter your contract here:". Below this, the contract text "and scale 5 one or one zoro" is visible. To the right of the input field are four blue buttons: "Help", "Input UNIX Time", "Evaluate Contract", and "Deploy Contract". Below the input field, there is a red error message box. The message is "Expected combinator, found: 'zoro'." and it is expanded to show a detailed trace: "At: 'zoro', atom: 6 of the contract.", "At: 'or', atom: 4 of the contract.", and "At: 'and', atom: 0 of the contract.".

Figure E.5: The *Composition* view, with an error message and expanded error trace.

Evaluating a SmartFin Financial Contract

In order to evaluate the SmartFin contract before deploying it, press the *Evaluate Contract* button. This will display the *Evaluate Contract* menu. For more information about this menu, please see section E.7.

E.5.2 Deploying a Financial Smart Contract

Once the SmartFin financial contract definition is fully-written and error-free, its financial smart contract representation can be deployed to the connected blockchain. The smart contract is compiled to the **wasm** format, so please ensure that the connected blockchain is compatible with **wasm** smart contracts before proceeding.

To open the deployment menu, press the *Deploy Contract* button on the composition view (or press **enter** in the SmartFin contract text input). This will display a short menu for setting some options on the contract (as shown in figure E.6).

A deployed financial smart contract has a counter-party (the account used to deploy the contract), and a prospective holder. The prospective holder of the financial smart contract is fixed, to ensure that only the intended party may acquire the contract. The holder's address must be provided to the contract during deployment, and can be entered in the *Contract Holder* input.

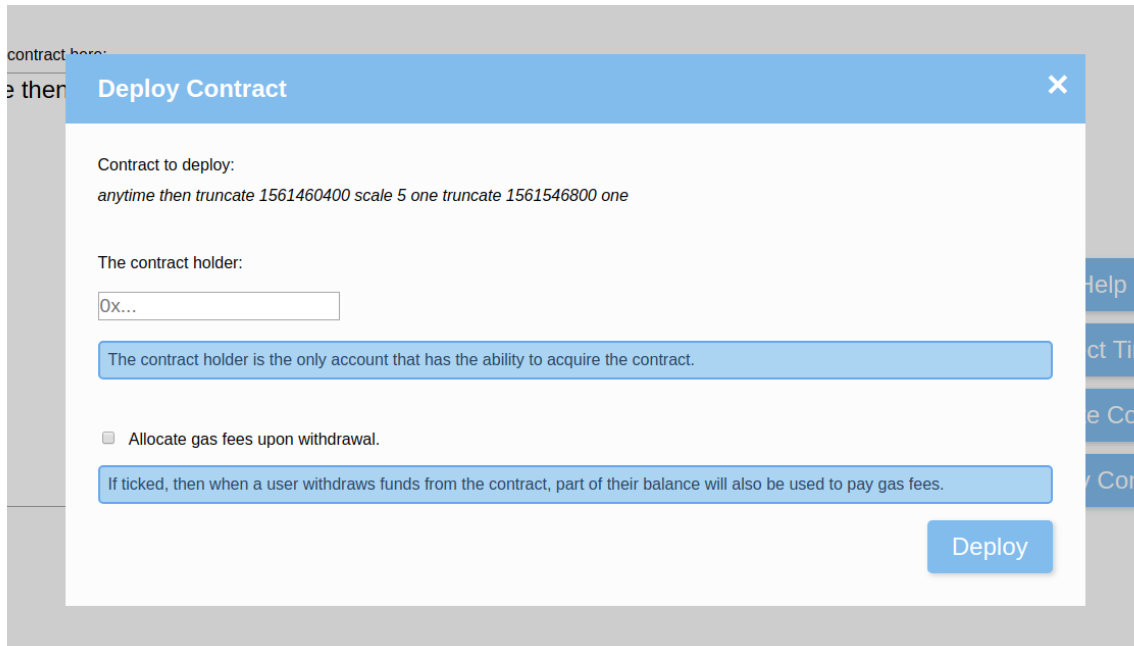


Figure E.6: The *Deploy Contract* menu, which displays the final SmartFin contract, as well as several options.

The financial smart contract also has an option for whether or not it should use gas upon withdrawal. If only externally-owned accounts (i.e. users, not other smart contracts) will interact with the contract, or the blockchain the contract is running on does not require gas fees, then this box does not need to be ticked. If one of the parties who will withdraw funds from the contract is a smart contract, then gas fees will need to be paid upon withdrawing in case the smart contract receiving the funds needs to execute code. In this case, the gas fees (2300 Wei) will be taken out of the withdrawing party's balance upon withdrawal. If there are not enough funds to pay for gas fees, then withdrawal will fail.

Once you have provided all options to the deployment menu, the *Deploy* button will deploy the financial smart contract to the connected blockchain from the connected account. If this fails for any reason, the error message will be displayed at the bottom of this menu. This message can be expanded by clicking to see the blockchain's error message.

E.6 Monitoring a Financial Smart Contract

From the main menu, you may have progressed to the monitoring view. If you progressed to the composition view instead, please see section E.5. The monitoring view enables you to interact with already deployed financial smart contracts in various ways.

If you have previously deployed a financial smart contract in the composition view during the current session on the client, then the monitoring view will automatically display the details of the deployed contract. If not, you will be prompted for the address of the financial smart contract you would like to monitor (as shown in figure E.7). If the given address corresponds to a financial smart contract then its details will be loaded, otherwise an error message will be shown.

E.6.1 Financial Smart Contract Details

The monitoring view allows the user to view several details from the financial smart contract, as shown in figure E.8. These details are contained with drop-down boxes, which can be opened

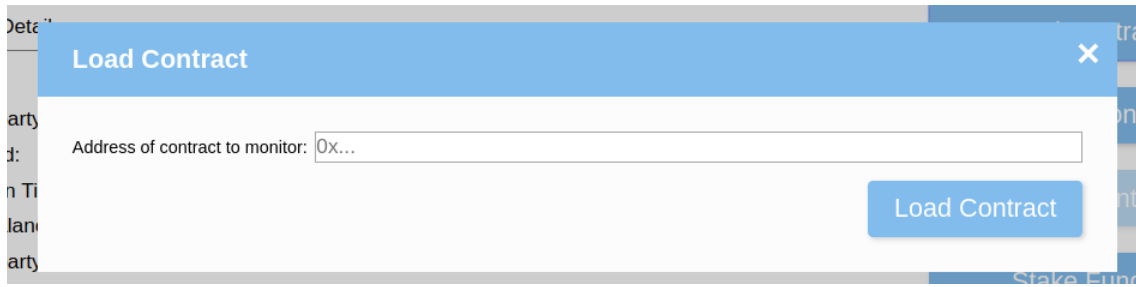


Figure E.7: The *Load Contract* menu, which allows you to enter the address of a financial smart contract you would like to monitor.

or closed by clicking on the titles. The *Contract Details* box contains information regarding the financial smart contract's current state, including the contract holder and counter-party's addresses, whether or not the contract has concluded, the top-level acquisition time, the holder and counter-party's balances, whether or not the contract uses gas, and the last time the contract was updated. The SmartFin financial contract definition for the financial smart contract is also displayed. All of the information in the monitoring view is refreshed every few seconds. The functions on the smart contract which return this information are *pure*, and so no gas is used to obtain these details.

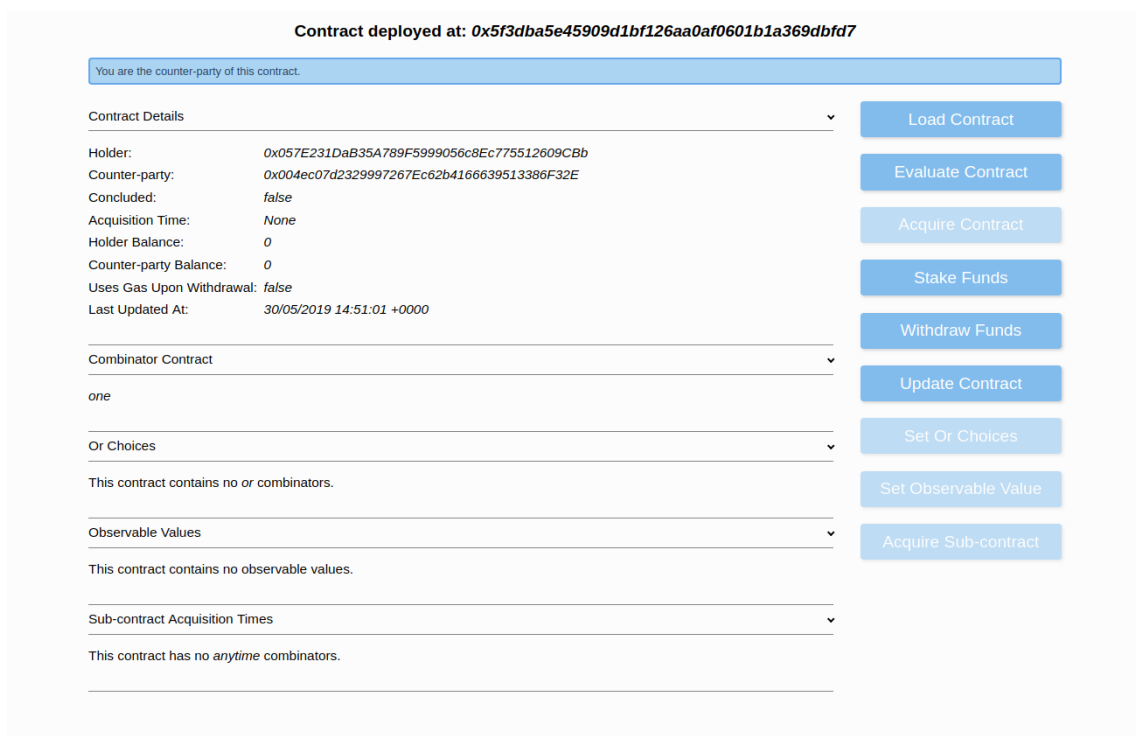


Figure E.8: The *Monitoring* menu, which displays the details of a financial smart contract in several drop-down boxes, and several options.

Besides the basic details, the contract also shows a list of available or-choices, observable values, and anytime acquisition times (and the state/value of each). This can be seen in figures E.9, E.10, and E.11.

E.6.2 Interacting with a Financial Smart Contract

Besides viewing the details of the financial smart contract, the monitoring view also allows the user to interact with it. The user can step through the contract to evaluate it based on certain

Combinator Contract	▼
<i>or one zero</i>	
Or Choices	▼
<i>or combinator 0: None</i>	

Figure E.9: An or-choice (before choosing) in the *Monitoring* menu.

Combinator Contract	▼
<i>scale observable0 <0x057E231DaB35A789F5999056c8Ec775512609CBb> one</i>	
Or Choices	◀
Observable Values	▼
observable0: <i>None</i>	Arbiter: <i>0x057E231DaB35A789F5999056c8Ec775512609CBb</i>

Figure E.10: An observable value named *observable0* (before setting) in the *Monitoring* menu.

parameters (see section E.7 for more details), acquire the contract (if the user is the contract holder), stake funds in the contract, withdraw funds from the contract, update the contract, or set values in the contract.

Acquiring and Updating

Acquiring the contract (i.e. *signing*) will set its acquisition time to the current time, and updates the contract based on its combinators. For example, if a user acquires the financial smart contract representing the SmartFin contract **one**, then the holder balance will increase by 1, and the counter-party balance will decrease by 1.

Some balance changes will only occur over time, depending on the combinators. For example, a **get** combinator will only acquire the sub-contract to update the contract balance upon its horizon, and before this its value is 0. The financial smart contract cannot automatically update over time, as this would be non-deterministic, and would require someone to pay gas fees. To bring the contract's value up-to-date, press the *Update Contract* button. *This requires paying gas, as updating requires state-altering execution on the smart contract.*

Staking and Withdrawing Funds

A financial smart contract cannot pay funds to either party until some funds are paid in. To put funds into the contract, press the *Stake Contract* button and enter a value (in Wei) to send. To withdraw funds, press the *Withdraw Funds* button and enter a value (in Wei) to withdraw. You cannot withdraw more funds than your balance, more funds than the total contract balance, or when your balance or the financial smart contract's total balance cannot afford the gas fees (when gas fees upon withdrawal are enabled).

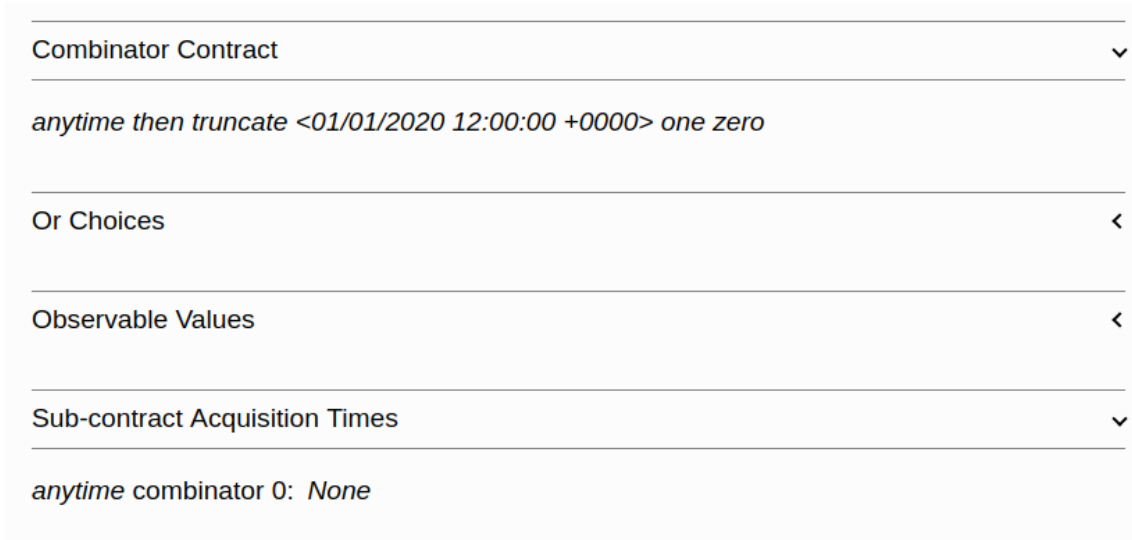


Figure E.11: An anytime acquisition time (before acquisition) in the *Monitoring* menu.

Choosing or Branches

You may also set several values on the financial smart contract. A contract with **or** combinators allows the user to choose between the left and right sub-contracts. To do this, you may press the *Set Or Choice* button and choose an **or** combinator and its chosen sub-contract (as shown in figure E.12). The **or** combinator is selected by its **or-index**. Each **or** combinator has an **or-index**, starting from zero and increasing sequentially by order of occurrence in the financial smart contract (from left to right). For example, in the contract **or one or one zero**, the **or** at atom 0 has **or-index** 0, and the **or** at atom 2 has **or-index** 1.

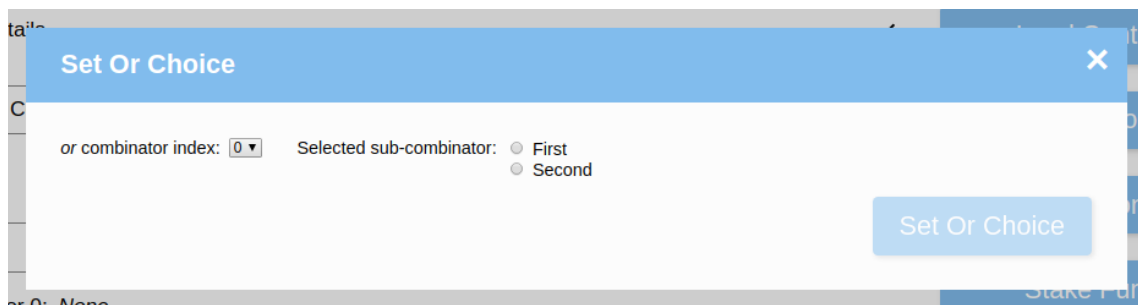


Figure E.12: The menu to choose a branch for an **or** combinator.

Setting Observable Values

From the monitoring view, you may also set the value of observables (as shown in figure E.13). The **scale** combinator may define an observable and an arbiter. If you are logged in to the client with the address of an arbiter of an observable, then you can set the value of that observable. To do so, press the *Set Observable Value* button, choose the observable to set, and enter the value.

Acquiring anytime Sub-contracts

In a financial smart contract, an **anytime** combinator can be acquired at any point before the horizon is passed. If it is not acquired by this time, it will be acquired on the horizon. For more details, see section E.2. In order to acquire an **anytime** sub-contract at the current time, press the *Acquire Anytime Sub-contract* button to open the **anytime** acquisition menu (shown in figure E.14). Choose the combinator by its **anytime-index** (defined similarly as the **or-index** in section

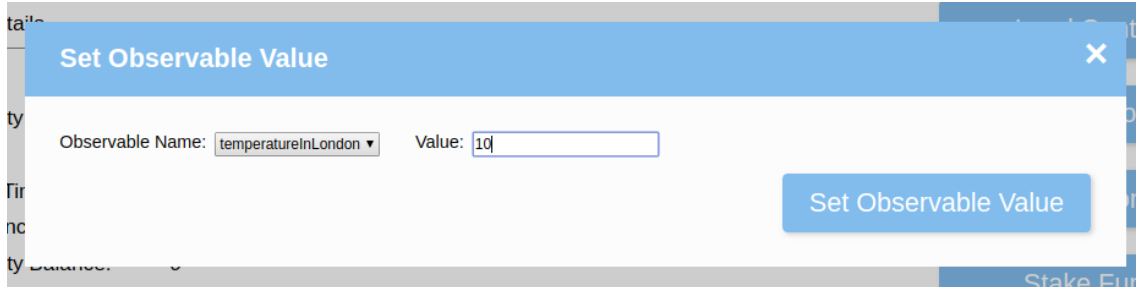


Figure E.13: The menu to set the value of an observable (named *temperatureInLondon*).

E.6.2, for *anytime* instead of *or*), and press the *Acquire* button. If you have not acquired the parent of the *anytime* combinator, then this will fail and display an error message.

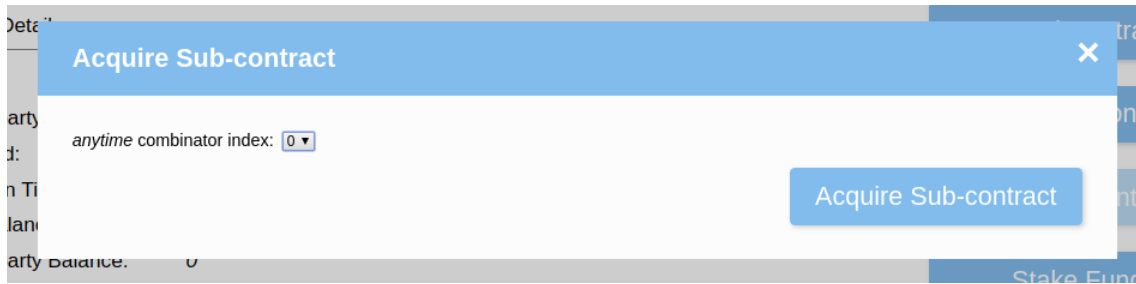


Figure E.14: The menu to acquire an *anytime* combinator.

E.7 Evaluating a SmartFin Financial Contract

The evaluation menu can be viewed from either the *composition* view or the *monitoring* view. The SmartFin financial contract which has been composed or loaded from a monitored financial smart contract can be evaluated in this view, by selecting acquisition times and or-choices for the contract. The evaluation is handled in a step-by-step manner, and at each step you are required to choose one of several options. These options can be deleted at any time, in case you change your mind. The first step of evaluation will be choosing the top-level contract acquisition time, i.e. the time the contract itself is acquired.

E.7.1 Top-level Acquisition Time

When choosing the top-level acquisition time, several options will be prevented to you (as shown in figure E.15). These options are each a distinct range of time, within which the contract's value will not change. For example, the contract `truncate <1st March 2020 12:00> one` will have the value 1 Wei until 12:00 on the 1st March 2020 is passed, and the value 0 Wei afterwards. This makes 2 distinct ranges of time with fixed values.

These time ranges do not account for the varying of observables; for example, `scale obs <addr> one` will be treated as a single range of time with equal value, even though the value of its observable may change over time.

E.7.2 Anytime Acquisition Time

Similarly to the top-level acquisition time, you may also need to provide acquisition times for anytime combinators within a contract, as shown in figure E.16. The options for this will also be represented as distinct time ranges, in the same way that the top-level acquisition time options

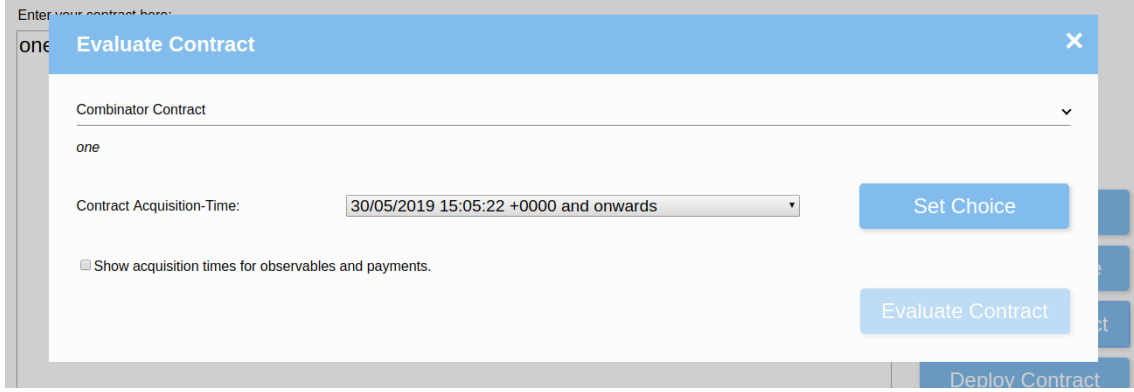


Figure E.15: The *Evaluation* menu, before choosing the contract's acquisition time.

are shown. For example, for the contract **anytime then truncate <1st March 2020 12:00:00> one zero** (with a top-level acquisition time of 1st March 2020 12:00:00 or earlier), **anytime 0** will have options consisting of the time before 12:00 on 1st March 2020, and the following time period.

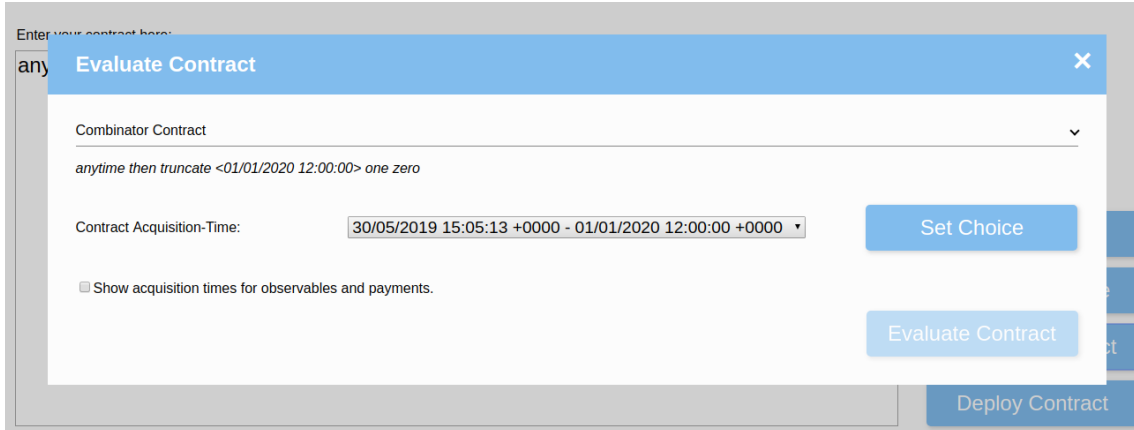


Figure E.16: The *Evaluation* menu, setting an **anytime** acquisition time.

If an **anytime** combinator only has one available option for its acquisition time, then that option will be chosen automatically. For example, in the same contract defined above, if the top-level acquisition time is after 12:00:00 on 1st March 2020, then **anytime 0** can only be acquired in the time range after this time, and so this is automatically set. As such, the user only needs to select **anytime** acquisition times where there is more than one time-range to choose from.

The **anytime** combinator is denoted by its **anytime-index**, which is defined similarly to the **or-index** in section [E.6.2](#).

E.7.3 Or Choices

Besides acquisition times, the other combinator which requires user input is the **or** combinator. As such, during step-by-step evaluation the user must input any **or-choices** as they occur, as shown in figure [E.17](#). You may choose between the first and second branch of the **or** combinator, ordered by occurrence when reading the SmartFin financial contract definition from left-to-right. For example, in the contract **or one zero**, choosing the first branch will net a value of 1 Wei, and the second branch will net a value of 0 Wei.

If either branch is expired at the time that the **or** combinator is encountered during evaluation (i.e. the time-range chosen for its parent combinators), then that branch cannot be acquired (for more information, see section [E.2](#)). For example, in the contract **or truncate <1st March 2020**

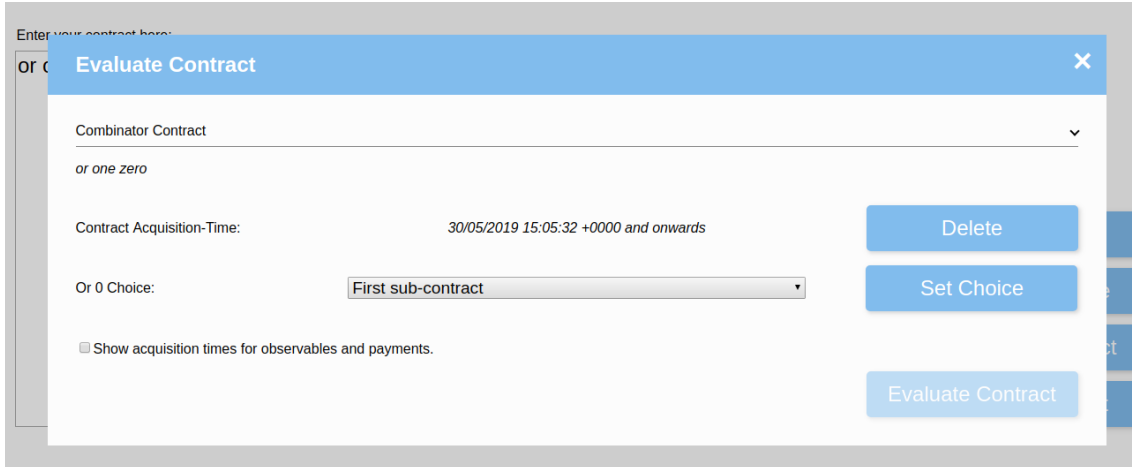


Figure E.17: The *Evaluation* menu, setting an *or* choice.

12:00:00> *one zero*, if the top-level contract is acquired after 12:00:00 on the 1st March 2020, then no *or* choice will be presented and the value will be 0 Wei. If both branches have expired, then similarly no choice is presented and the *or* combinator will evaluate to 0 Wei.

The *or* combinator is denoted by its *or-index*, which is defined in section E.6.2.

E.7.4 Value

The final result of the evaluation process will be a value, displayed at the bottom of the menu (as shown in figure E.18). This value consists of a sum of products of observables and values. For example, the contract *and scale obs0 <addr> one scale obs1 <addr> one* will evaluate to $obs0 * 1 + obs1 * 1$ Wei. The value of the observable cannot be estimated by the evaluation process, and so it is treated as an unknown variable.

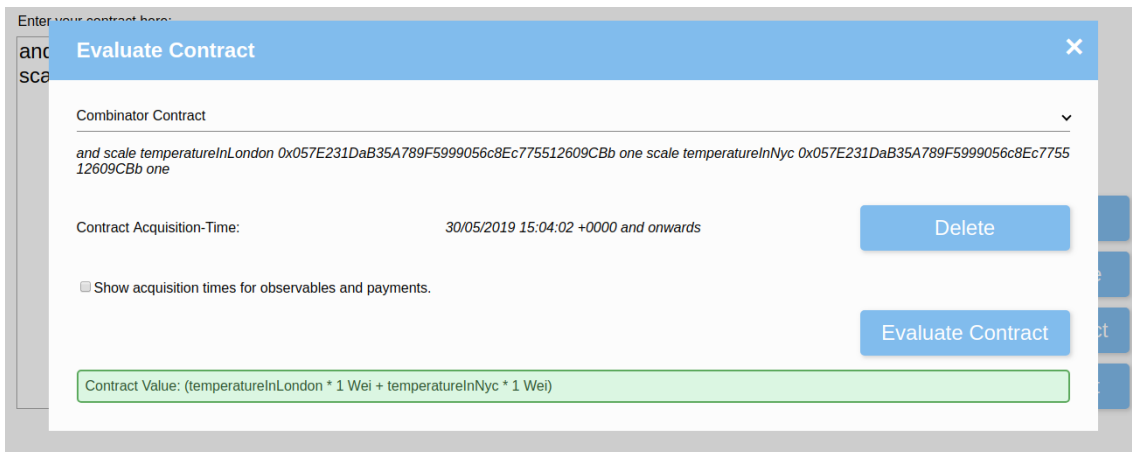


Figure E.18: The *Evaluation* menu, after evaluating a SmartFin financial contract.

By default, the evaluation value represents the total overall value of the contract once it has concluded, and does not take time into account. Some observables may have widely varying values over time, in which case it can be useful to know the specific times in which observables/payments are acquired. In order to show this, check the box labelled "*show acquisition times for observables and payments.*" This will display time periods alongside observables and payment values, which represent the times at which they are each queried or obtained respectively (as shown in figure E.19).

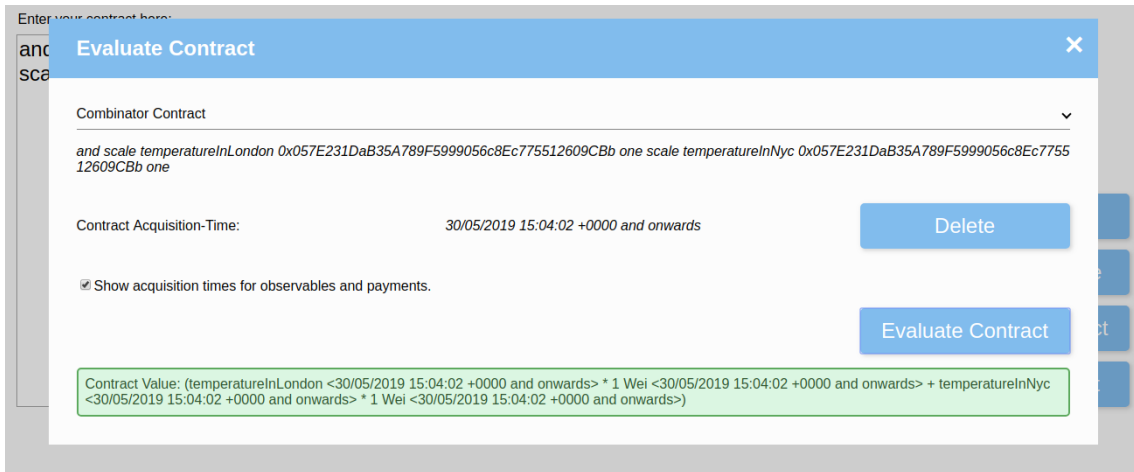


Figure E.19: The *Evaluation* menu, after evaluating a SmartFin financial contract with acquisition times displayed.

Bibliography

- [1] Steve Akinyemi. Awesome WebAssembly Languages. <https://github.com/appcypher/awesome-wasm-langs>. (Accessed: 21-01-2019).
- [2] Babel. Babel · The compiler for next generation JavaScript. <https://babeljs.io/>. (Accessed: 14/06/2019).
- [3] Browserify. Browserify. <http://browserify.org>. (Accessed: 14/06/2019).
- [4] Hampton Catlin, Natalie Weizenbaum, Chris Eppstein, Jina Anne, and numerous contributors. Sass: Syntactically Awesome Style Sheets. <https://sass-lang.com/>. (Accessed: 14/06/2019).
- [5] CoinGecko. Ethereum USD Chart (ETH/USD) | CoinGecko. <https://www.coingecko.com/en/coins/ethereum/usd>. (Accessed: 13/06/2019).
- [6] Coinmonks and G Nicholas D'Andrea. EthPM: Reusable smart contract packages. <https://medium.com/coinmonks/ethpm-smart-contract-packages-for-developers-81c77481c491>. (Accessed: 13-01-2019).
- [7] ConsenSys. Truffle Overview. <https://truffleframework.com/docs/truffle/overview>. (Accessed: 15-01-2019).
- [8] Yarn Contributors. Yarn. <https://yarnpkg.com/en/>. (Accessed: 14/06/2019).
- [9] DFINITY. Dfinity Haskell Compiler. <https://github.com/dfinity/dhc>. (Accessed: 21-01-2019).
- [10] ConsenSys Diligence. Known Attacks - Ethereum Smart Contract Best Practices. https://consensys.github.io/smart-contract-best-practices/known_attacks/#reentrancy-on-a-single-function. (Accessed: 13/06/2019).
- [11] Jordan Earls. The Faults and Shortcomings of the EVM. <https://medium.com/@earlz/the-faults-and-shortcomings-of-the-evm-bde4d09b8b6a>. (Accessed: 21-01-2019).
- [12] Ethereum Foundation. Contracts - Solidity 0.4.24 documentation. <https://solidity.readthedocs.io/en/v0.4.24/contracts.html>. (Accessed: 13-01-2019).
- [13] Ethereum Foundation. Go-Ethereum Wiki. <https://github.com/ethereum/go-ethereum/wiki/Private-network>. (Accessed: 15-01-2019).
- [14] Ethereum Foundation. Remix, Ethereum-IDE 1 documentation. <https://remix.readthedocs.io/en/latest/>. (Accessed: 21-01-2019).
- [15] Ethereum Foundation. Remix-Tests. <https://github.com/ethereum/remix/tree/master/remix-tests>. (Accessed: 21-01-2019).
- [16] Ethereum Foundation. web3.js - Ethereum JavaScript API - web3.js 1.0.0 documentation. <https://web3js.readthedocs.io/en/1.0/>. (Accessed: 07/06/2019).
- [17] Ethereum Foundation. web3.py - Web3.py 4.9.2 documentation. <https://web3py.readthedocs.io/en/stable/>. (Accessed: 07/06/2019).

- [18] Ethereum Foundation. What is Ethereum? - Ethereum Homestead 0.1 documentation. <http://www.ethdocs.org/en/latest/introduction/what-is-ethereum.html>. (Accessed: 13-01-2019).
- [19] Ethereum Foundation. What is Ethereum flavored WebAssembly (ewasm)? <https://ewasm.readthedocs.io/en/mkdocs/README/#what-is-ethereum-flavored-webassembly-ewasm>. (Accessed: 21-01-2019).
- [20] Ethereum Foundation. Yul - Solidity 0.5.9 documentation. <https://solidity.readthedocs.io/en/v0.5.9/yul.html>. (Accessed: 11/06/2019).
- [21] JS Foundation. Mocha - the fun, simple, flexible JavaScript test framework. <https://mochajs.org/>. (Accessed: 14/06/2019).
- [22] JS Foundation. webpack. <https://webpack.js.org/>. (Accessed: 14/06/2019).
- [23] Python Software Foundation. Welcome to Python.org. <https://www.python.org/>. (Accessed: 14/06/2019).
- [24] Facebook Inc. React - A JavaScript library for building user interfaces. <https://reactjs.org/>. (Accessed: 14/06/2019).
- [25] Google Inc. Google Chrome - The Fast, Simple and Secure Browser from Google. <https://www.google.com/chrome/>. (Accessed: 14/06/2019).
- [26] Tweag I/O. Asterius: A Haskell to WebAssembly Compiler. <https://github.com/tweag/asterius>. (Accessed: 21-01-2019).
- [27] Simon P. Jones, Jean-Marc Eber, and Julian Seward. Composing Contracts: An Adventure in Financial Engineering. *ACM SIG-PLAN Notices*, 35(9):280–292, 2000. <https://www.cs.tufts.edu/~nr/cs257/archive/simon-peyton-jones/contracts.pdf> (Accessed: 13-01-2019).
- [28] Will Kenton. Lattice-Based Model. <https://www.investopedia.com/terms/l/lattice-model.asp>. (Accessed: 13/06/2019).
- [29] Igor Korsakov. Awesome Smart Contracts. <https://github.com/Overtorment/awesome-smart-contracts>. (Accessed: 24/01/2019).
- [30] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. *ACM CCS 2016*, pages 254–269, 2016. <https://eprint.iacr.org/2016/633> (Accessed: 15-01-2019).
- [31] MetaMask. MetaMask - About. <https://metamask.io/>. (Accessed: 21-01-2019).
- [32] Moment. Moment.js | Home. <https://momentjs.com/>. (Accessed: 14/06/2019).
- [33] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. <https://bitcoin.org/bitcoin.pdf>. (Accessed: 15-01-2019).
- [34] Valérian Rousset. WebAssembly for Scala. <https://github.com/tharvik/wasm>. (Accessed: 21-01-2019).
- [35] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. Writing safe smart contracts in Flint. *Programming'18 Companion*, pages 218–219, 2018. <https://dl.acm.org/citation.cfm?doid=3191697.3213790> (Accessed: 15-06-2019).
- [36] Rust Team. Cargo: packages for Rust. <https://crates.io/>. (Accessed: 14/06/2019).
- [37] Rust Team. Rust Programming Language. <https://www.rust-lang.org/>. (Accessed: 14/06/2019).
- [38] Parity Technologies. pwasm-test. <https://github.com/paritytech/pwasm-test>. (Accessed: 07/06/2019).
- [39] Parity Technologies. pwasm_abi - Rust. https://paritytech.github.io/pwasm-abi/pwasm_abi/. (Accessed: 07/06/2019).

- [40] Parity Technologies. pwasm_ethereum - Rust. https://paritytech.github.io/pwasm-ethereum/pwasm_ethereum/. (Accessed: 07/06/2019).
- [41] Parity Technologies. pwasm_std - Rust. https://paritytech.github.io/pwasm-ethereum/pwasm_std/. (Accessed: 07/06/2019).
- [42] Parity Technologies. A step-by-step tutorial on how to write contracts in Wasm for Kovan. <https://github.com/paritytech/pwasm-tutorial>. (Accessed: 21-01-2019).
- [43] Parity Technologies. A step-by-step tutorial on how to write contracts in Wasm for Kovan - Run node and deploy contract. <https://github.com/paritytech/pwasm-tutorial#run-node-and-deploy-contract>. (Accessed: 31/05/2019).
- [44] Parity Technologies. Technologies. <https://paritytech.github.io/technologies.html>. (Accessed: 07/06/2019).
- [45] Parity Technologies. wasm-utils. <https://github.com/paritytech/wasm-utils#build-tools-for-cargo>. (Accessed: 31/02/2019).
- [46] Parity Technologies. WebAssembly (Wasm) · Parity Tech Documentation. <https://wiki.parity.io/WebAssembly-Home>. (Accessed: 07/06/2019).
- [47] W3C. Current Members - W3C. <https://www.w3.org/Consortium/Member/List>. (Accessed: 21-01-2019).
- [48] W3C. Standards - W3C. <https://www.w3.org/standards/>. (Accessed: 54/06/2019).