Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

Free Trade: Composable Smart Contracts

Ross Gardiner

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

15th May 2018

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Ross Gardiner, 15th May 2018

Contents

Executive Summary

My research hypothesis is that a modern reimplementation of ‘composing contracts’, initially described by Peyton Jones et al. [1], can be used as a declarative language for the creation of Ethereum ‘smart contracts’. During the project:

- I learned about:
  - the composing contracts DSL and its valuation semantics;
  - the free monad, interpreters and their implementation in Haskell;
  - the ‘Data Types à la Carte‘ approach to modular data types and interpreters
  - and authoring smart contracts and Đapps for Ethereum.
- I reimplemented the composing contracts DSL in a free monadic style, demonstrating the use of the techniques above. I also:
  - modularised the language and demonstrated how interpreters can be defined on different subsets
  - and implemented a number of simple demonstration interpreters, including rendering to a graph and valuation
- I built a system for deploying contracts written in the composing contracts DSL as Ethereum smart contracts. To do this, I:
  - unified the contract primitives proposed in the two composing contracts papers;
  - implemented an interpreter to compile the DSL to Solidity, the most popular language for writing smart contracts (approx. 2000 lines of Haskell);
  - iterated the compiler with basic efficiency improvements;
  - and implemented an Ethereum ‘Đapp’ (distributed app) to deploy and manage compiled contracts (approx. 1500 lines of ECMAScript, TypeScript and Vue).

Supporting Technologies

- I used the *Haskell* language (GHC 8.2.2, Stack 1.7.1). I also used a number of Haskell libraries, including:
  - Edward Kmett's *free* [2] and *lens*
  - Matthew Sackman and Ivan Lazar Miljenovic's *graphviz* [3]
  - and others (see `merchant.cabal`)
- I wrote the Đapp in a mix of *TypeScript* and *ECMAScript*. It was built with the *Vue.js* [4] and *web3.js* [5] libraries.
- I used a mixture of *go-ethereum* (geth), *Parity*, *Mist*, *Remix* and *MetaMask* to simulate and interact with an Ethereum blockchain.

Acknowledgements

Many thanks to my supervisor, Dr. Nicolas Wu, for his advice, and for always being two steps ahead.

I would also like to acknowledge the irony that I chose this project because I am interested in strongly-typed functional programming, but then ended up writing over 1000 lines of JavaScript.

Chapter 1

Introduction

Since the 1970s, financial markets have been rapidly digitised. At the same time, trading volumes have vastly increased. Looking forward, as internet connectivity begins to permeate everything, we are likely to see another shift towards market-based allocation and control of resources (e.g. storage, compute, physical objects) [6]. As well as increasing trading volumes by an order of magnitude, this will make market availability a critical requirement for everyday life.

An open question remains: are today's centralised markets suitable for this future? The high-level goal of this project is to lay the foundations for an alternative financial derivatives market—one which is robust, distributed and highly-automated. This would have numerous potential benefits:

Reliability Centralised exchanges have suffered downtime in the past [7], disrupting financial markets. A distributed platform eliminates some of these single-point-of-failure issues.

Speed Settlement is the process of exchanging financial assets in order to fulfil a contract. In the past, this meant delivering physical paper certificates or forms. In modern markets, these are normally held by a central depository and transfer of ownership simply involves updating a database record. However, despite this, many markets operate on a T+2 settlement cycle [8], [9]—meaning that this process can take up to two days. A more automated approach to describing financial derivatives should make 'straight-through processing'—immediate processing of transactions with no human intervention—the norm.

Transparency Currently, markets can operate with very little transparency. In particular, exotic (non-standard) financial products are normally traded *over-the-counter*. This means that transactions are peer-to-peer and private. A system that allowed these products to be traded on a standard exchange would encourage a more open marketplace.

To achieve these goals, I will build on two key existing pieces of technology: the *composing contracts* language and Ethereum.

1.1 Composing contracts

Composing contracts is a domain-specific language (DSL) for describing financial contracts. Other examples of domain-specific languages include Markdown (for formatting text), Blueprints (for game logic in Unreal Engine) and Cucumber (for writing human-readable automated tests). A well designed DSL can have several benefits compared to a general-purpose programming language. These include:

- clear, declarative code that maps directly to the problem domain;
- increased safety through making invalid operations and states unrepresentable;
- code that can be understood by domain experts who are not programmers;
- and increased potential for optimisation because code represents semantic intention rather than comprising a sequence of general operators.

Proposed by Simon Peyton Jones, Jean-Marc Eber and Julian Seward in 2000 [1], the DSL can be used to precisely describe the semantics of complex financial derivatives. This makes it possible to build systems that 'understand' the semantics of a contract, and which can perform activities like contract valuation or automated trade execution.

Crucially, the particular design of this DSL makes it trivial to construct new types of financial contract from a toolbox of well-understood primitives. It has proven useful enough to be commercialised by LexiFi SAS in its *MLFi* product [10].

1.2 Ethereum

Ethereum is a blockchain platform that allows distributed execution of programs (*smart contracts*). Ethereum has many similarities to Bitcoin, a decentralised digital currency. Bitcoin pioneered blockchain as a method of maintaining distributed consensus over the state of its payments ledger.

In addition to providing a digital currency (known as Ether), Ethereum allows users to store smart contracts in its blockchain ledger. Ethereum users can call methods exposed by these smart contracts, with the blockchain storing the transaction and the resulting state updates. These smart contracts can represent structures including voting systems, auctions and escrow services. Smart contracts have two key features: everyone can see how they will execute, and they are immutable once deployed. This makes it possible for all participants to independently verify that a system is fair.

1.3 Aims

The primary goals of this project are to:

- **reimplement composing contracts using modern Haskell** techniques including the free monad and 'Data Types á la Carte' [11]-style modularity;
- **adapt the language** as necessary to match the semantics of Ethereum;
- **write a compiler** that converts the composing contracts language to Ethereum smart contracts;
- **build a prototype client** for deploying and managing contracts;
- **evaluate the usability** of composing contracts for authoring Ethereum smart contracts;
- and explore methods for **reducing the execution cost** of compiled contracts.

1.4 Overview

Chapter 2 explains the general social and technical background to this project. Here I give an overview of financial markets, including the design and purpose of various types of financial instrument. I then describe how many such contracts can be expressed in the composing contracts language. In addition, I explain the history and design of Ethereum, including a simple introduction to writing smart contracts.

Chapter 3 walks through, in detail, the design decisions made during the implementation of the contract compiler. I compare various approaches to embedding DSLs in Haskell, then explain `Fix` and `Free` and show how they can be used to implement a modular DSL. Finally, I describe the process of compiling contracts to Solidity, executing and managing them.

Chapter 4 explores whether my implementation achieves the technical goals outlined above, and considers the impact of several implementation choices. I compare my implementation to others and measure the relative cost of deploying and executing contracts.

Chapter 5 sums up the findings of the project. I explore whether we have moved any closer to the goal of reliable, fast and transparent markets.

Chapter 2

Background

2.1 Financial Markets

The idea of a financial contract is surprisingly ancient. In the 4th century BCE, Aristotle appears to describe a form of option contract in *Politics* [12]. Thales the Milesian, he says, paid a small deposit ahead of the harvest to secure all of the olive presses in the region. Having a monopoly, he was able to sublet them at a high price when the harvest arrived.

Over thousands of years, a deep and complex market of financial instruments has evolved. An *instrument* is any contract that 'gives rise to a financial asset of one entity and a financial liability [...] of another entity' [13]. These contracts may or may not be tradeable: a *security* is something that can be traded. Instruments like loans cannot be traded without mutual agreement between the lender and borrower, but there has been a recent trend towards *securitisation*—the reselling, as tradeable securities, of cashflows from debt repayment.

Initially, stocks and other financial instruments were traded ad-hoc or through informal venues like coffee shops. The turn of the 19th century saw the establishment of the stock exchanges that we are familiar with today. The New York Stock Exchange and London Stock Exchange started in 1792 and 1801 respectively. The operations of these exchanges remained almost entirely paper-based until the market was disrupted by the creation of NASDAQ in 1971. The world's first electronic exchange, NASDAQ is now the world's second largest exchange by market capitalisation. Competitors were forced to modernise, and now fully-automated stock exchanges are the norm.

There are some key categories of financial instrument, which I will describe briefly here.

2.1.1 Debt

Debt is an obligation to pay or be paid by another party. It comes in many forms, including bonds, loans and mortgages. We are principally interested in debt *securities*, because they are tradeable—often standardised—assets.

Bonds are one such security and provide an alternative to loaning money from a single entity. Entities such as corporations, government or universities wishing to raise money can *issue* bonds with a fixed *face value*. Investors that purchase these bonds will receive interest (at the *coupon rate*) on the face value of the bond each year. At the *maturity date* of the bond (when it expires) the issuer pays the bond holder the face value of the bond.

Being securities, bonds do not have to be kept by the original purchaser and can be traded on the bond markets. Bonds often have a lifespan of 30–50 years, during which time interest rates can fluctuate significantly in the rest of the market. As a result, the cost of purchasing a bond may be different from its face value. *Premium* bonds are those that have a market price higher than face value; *discount* bounds have a price lower than face value.

2.1.2 Derivatives

Derivatives are financial instruments whose value depends on one or more other instruments or assets. There are a few common types of derivative, but a vast array of subtypes and specialised instruments.

Forwards and futures

Futures are financial contracts that oblige the holder to either buy or sell something at a predetermined price and time. This is useful because it allows you to hedge risk. For example, if you are a manufacturer and know that you will need a particular commodity (e.g. steel) in a year's time, you can buy a steel future. This means that you can run your business knowing the exact amount of money you will need in a year's time. If the price of steel increases unexpectedly, you will be unaffected. If it goes down unexpectedly, however, you will be obliged to overpay.

The Dojima Rice Exchange in early 18th century Japan is recognised as the first futures trading venue. Though futures trading was prohibited by the shogunate, it was tacitly permitted when the price of rice was thought to be too low [14].

A *forward* is simply a non-standardised futures contract.

Options

Options are similar to futures—they allow the holder to buy or sell something (*call* and *put* options, respectively) at a predetermined price and time. Unlike a future, however, an option gives you a *choice* to either purchase the underlying contract or do nothing. As a result of this increased flexibility, it costs a small *premium* to buy these contracts.

Swaps

Swaps are very general, simply being the exchange of two financial instruments. The most common type is the *interest rate swap*. This allows two entities to swap cashflows arising from an interest rate. For example, two companies might issue bonds—one with a fixed interest rate, the other with a variable interest rate related to the LIBOR rate[1]. Depending on how the entities believe LIBOR will change, they may decide to enter into a *swap* of their interest obligations.

2.1.3 Stocks

Ownership of *stock* represents fractional ownership of a corporation. The holder of the stock thus has some claim to the assets and earnings of that corporation.

The first company to publicly offer shares was the Dutch East India Company (VOC). It created the Amsterdam Stock Exchange to provide a venue for the trading of both its stock and the bonds that it issued to finance voyages.

2.2 Describing Contracts

It is very useful to have a programmatic representation of financial contracts. This has many uses:

Precision Programmatic representations provide an unambiguous electronic record of the contracts you have bought or sold. The representation encodes the semantics of the contract, rather than a human-readable description which may be interpreted in multiple ways.

Automation Trading and settlement of contracts can be automated if the behaviour can be executed by a computer. Removing the human from this process considerably reduces the cost of handling financial contracts.

Risk management Banks and other trading entities can get real-time insight into the risks present in their portfolio and order book if these are represented electronically.

Valuation Given a programmatic representation of a contract, traders and auditors can use computational modelling tools to automatically value the contract.

There are many approaches to solving this problem. Traditionally, contracts have been represented in a procedural or object-oriented style. Considerable development work is required to create new types of contract and it is somewhat difficult to introspect a contract's inner workings. Other approaches include logic programming [15] and custom languages (e.g. RISLA [16]).

[1] LIBOR is the *London Inter-bank Offered Rate*. This is the average interest rate at which some of the major London-based banks can take a short-term loan.

We will take a different approach, optimising for the composability and semantic understandability of contracts. To this end, we will use a combinator language proposed by Peyton Jones et al. in their 2000 paper *Composing contracts: an adventure in financial engineering* [1]. Because this language forms the basis of everything in this project, we will spend some time exploring it in detail.

For example, first imagine that we want to describe a *zero-coupon discount bond.* We've already seen what a bond is, but let's break down the other terms. A zero-coupon bond does not pay any interest. A discount bond is sold below its face value. In fact, in this (unrealistic) case, it is being given away for free.

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

We are using Haskell as the implementation language for our contract language. The syntax `c1 :: Contract` indicates that `c1` has type `Contract`, and `c1 = zcb t1 100 GBP` defines the value of `c1` as `zcb t1 100 GBP`. Note that function application in Haskell is represented by whitespace, so this is roughly equivalent to `zcb(t1, 100, GBP)` in most other languages.

We use `zcb` to describe a zero-coupon bond that matures at time `t1` with a face value of 100 GBP. In fact, `zcb` is a function of type:

```haskell
zcb :: Date → Double → Currency → Contract
```

We can now imagine combining two contracts together. For example, to combine the effect of two contracts we can use the `and` combinator:

```haskell
and :: Contract → Contract → Contract
```

```haskell
c2, c3 :: Contract
c2 = zcb t2 200 GBP
c3 = c1 `and` c2
```

The contract `c3` causes the holder to receive 100 GBP at `t1` *and* 200 GBP at `t2`. Note the use of backticks (`) to turn `and` into an infix function—that is, ``` `and` ``` appears *between* its arguments. Normally, functions are written in prefix notation (`and c1 c2`).

Contracts are a mutual agreement between two parties, whom we refer to as the `holder` and the `counterparty`. In general, the counterparty is the entity that sells the contract. The holder is the purchaser of the contract and makes any necessary choices during execution (e.g. for an option contract).

We can swap the rights and obligations conferred by a contract using the `give` combinator.

```haskell
c4 :: Contract
c4 = c1 `and` give c2
```

While `c3` represents the holder receiving 100 GBP at `t1` and 200 GBP at `t2`, instead `c4` represents receiving 100 GBP at `t1` and paying 200 GBP to the counter-party at `t2`.

Our contract semantics are defined by the idea of *acquisition.* Upon acquiring a contract, the parties may be conferred some right or obligation. For example, `one USD` obliges the counterparty to immediately give one USD to the holder. A contract may also permit (or force) the holder to acquire another 'underlying' contract at some future time. Every contract has an expiry time (*horizon*), beyond which it can no longer be acquired. This expiry time may be finite or infinite.

It is also worth noting that our contract definitions are *descriptions* of a single transaction between two pre-defined parties. They are not objects that can themselves be arbitrarily exchanged between entities.

2.2.1 Breaking down the zero-coupon bond

It turns out that our zero-coupon discount bond can be decomposed into a simpler set of primitives. Perhaps the simplest is `one`. When `one k` is acquired, the holder immediately receives one unit of currency `k` from the counter-party.

```haskell
one :: Currency → Contract
```

```haskell
c5 :: Contract
c5 = one GBP
```

Our contract `c5` has an infinite horizon—it can always be acquired. However, the payment is *immediate* upon acquisition of the contract and cannot be delayed.

In order to ensure that our payment occurs at precisely `t1`, we must control the acquisition date of the contract. To do this, we introduce two new combinators, `get` and `truncate`.

```
truncate :: Date → Contract → Contract
get :: Contract → Contract
```

The effect of `truncate t c` is just to wrap the underlying contract `c` in a contract with a horizon at time `t`. This means that `c` can no longer be acquired after `t`. The effect of `get c` is to force the underlying contract `c` to be acquired exactly at its horizon.

```
c6 = get (truncate t1 (one GBP))
```

In `c6`, we combine the `get` and `truncate` combinators to create a `one GBP` contract which must be acquired at `t1`. This is done by trimming the horizon to `t1` with `truncate`, then using `get` to force acquisition at that horizon.

However, we want to pay (say) 100 GBP, not 1 GBP. We could use `and` to combine together many `one`s, but this is rather impractical and does not work for fractional amounts. Instead we will introduce another combinator:

```
scaleK :: Double → Contract → Contract
```

When you acquire `scaleK x c`, you immediately acquire `c` but with all obligations (payments and receipts) scaled by `x`. Now we can define our `zcb` combinator in full:

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

In our definition of `zcb` we truncate the horizon to `t` and use `get` to force acquisition at the horizon. Then we scale the `one k` payment by a constant factor of `x`.

2.2.2 Observable values

An observable value is a quantity that can be measured at a particular time and date. Examples might include 'the current temperature in Bristol' or 'the current value of the CBOE Volatility Index (VIX)[2]'. These are useful for derivatives contracts. For example, you may hedge against volatility by scaling a contract relative to VIX, or hedge risk in a future for a food crop based on the weather.

We define the data type `Obs d` to represent an observable quantity of type `d`. This allows us to break down our `scaleK` combinator yet further. We can define a `scale` combinator which scales a contract dynamically by an observable factor:

```
scale :: Obs Double → Contract → Contract
```

Note the use of a type parameter in `Obs Double`. Much like generics in C♯ or Java, we could potentially create values of type `Obs` *anything*. We can now, for example, scale a contract by some `volatilityIndex`:

```
c7 :: Currency → Contract
c7 k = scale volatilityIndex (one k)
```

`scale` doesn't completely solve our problem. We need to scale by a fixed factor, but `scale` scales by an observable value, which might change over time. To solve this, we define a *constant* observable—i.e. one that is the same at all times.

```
konst :: a → Obs a
```

With this, we can now define `scaleK` in terms of `scale`:

```
scaleK :: Double → Contract → Contract
scaleK x c = scale (konst x) c
```

A number of useful observables can be constructed. `date :: Obs Date`, for example, simply returns the current date. We can even consider lifting arbitrary functions into the observable:

```
lift :: (a → b) → Obs a → Obs b
lift2 :: (a → b → c) → Obs a → Obs b → Obs c
```

Notice that `lift` takes a *function* as one of its parameters. This is a widely used pattern in Haskell, and is now becoming popular in other languages. In this case, we are effectively using `lift`/`lift2` to apply the provided function to the observable(s) at all points in time. If you are familiar with Haskell, you may notice that `lift` is analogous to `fmap :: Functor f ⇒ (a → b) → f a → f b`, and `lift2` to `liftA2 :: Applicative f ⇒ (a → b → c) → f a → f b → fc`.

There are some obvious applications of these combinators—for example, an `Obs Bool` that becomes true on a certain date:

[2] VIX—sometimes called the 'fear index'—predicts the amount of price volatility in S&P 500 stock prices over the next year.

```
at :: Date → Obs Bool
at t = lift2 (==) date (konst t)
```

In this case, we are using `date`, the observable that always returns the current date. We can even use several `lift2`s to define an observable that is true *between* certain dates.

```
between :: Date → Date → Obs Bool
between t1 t2 = lift2 (&&) (lift2 (≥) date (konst t1)) (lift2 (≤) date (konst t2))
```

Notice that we can have arbitrarily deeply nested observables if our contract calls for it.

2.2.3 Options contracts

Now we can think about some more complex types of contract. A *European option* is one that allows the holder to choose whether to acquire an underlying contract at an instantaneous moment in time. Here's how we could implement it:

```
european :: Date → Contract → Contract
european t u = get (truncate t (u `or` zero))
```

This is quite simple—we use the same `get`/`truncate` structure as before, but this time the holder is given a choice to either acquire the underlying contract `u` or acquire the null contract `zero` instead. `zero` is acquired immediately and has no effect—it acts as a kind of escape clause from the contract.

An *American option* allows the option to be taken at any point during a specified time interval.

```
american :: (Date, Date) → Contract → Contract
american (t1,t2) u = get (truncate t1 opt) `then` opt
  where opt :: Contract
        opt = anytime (truncate t2 (u `or` zero))
```

This is somewhat more complex to express. First we define `opt` to be an option that says 'the holder can acquire either `u` or `zero` at any time between acquisition of `opt` and `t2`'. The first operand of `then` expires at `t1`, so if the American option is acquired before or at `t1` the holder will acquire `opt` at `t1`. If the American option is acquired after `t1`, the holder will acquire the second operand of `then`, which is the option itself (exercisable any time up until `t2`).

We can simplify this by defining our own compound combinators:

```
at :: Date → Contract → Contract
at t c = get (truncate t c)

optionalUntil :: Date → Contract → Contract
optionalUntil t c = anytime (truncate t2 (c `or` zero))

american (t1,t2) u = at t1 opt `then` opt
  where opt :: Contract
        opt = optionalUntil t2 u
```

Notice that the `american` function takes a value of type `(Date, Date)`: this is a tuple of two date values. This function also uses a `where` clause to define additional values (i.e. `opt`) that are only accessible within the scope of the function.

The complete list of primitives proposed by the paper appears in Figure 2.1.

2.2.4 Rethinking the language

It's fair to say that this formulation of the American option is still rather unwieldy. Why should we have to use `then` and multiple `truncate`s to describe a seemingly simple concept? This problem may have inspired a reworked version of the original *Composing contracts* paper as a chapter of *The Fun of Programming* in 2003 [17].

In the updated paper, Peyton Jones and Eber replace many of the existing combinators (`then`, `truncate`, `get`, `anytime`). In fact, they do away with the notion of a contract's *horizon* entirely and use boolean observables to replicate its functionality. The replacement combinators are shown in Figure 2.2. Now we can look at how the contracts we have already defined might be re-expressed in this updated language.

The zero-coupon bond changes a little:

```
zcb :: Date → Double → Currency → Contract
zcb t x k = when (at t) (scaleK x (one k))
```

`zero :: Contract`
The null contract, `zero` confers no rights or obligations.
Horizon: Infinite
Acquisition: Acquired immediately

`one :: Currency → Contract`
The contract `one k` immediately pays the holder one unit of currency `k` upon acquisition.
Horizon: Infinite
Acquisition: Immediately acquire currency

`give :: Contract → Contract`
Acquiring `give c` entails immediately acquiring all the obligations of `c` as rights, and all the rights as obligations.
Horizon: Horizon of `c`
Acquisition: Immediately acquire inverted `c`

`and :: Contract → Contract → Contract`
Acquiring ``c1 `and` c2`` means immediately acquiring both `c1` and `c2`. One contract being expired does not affect acquisition of the other.
Horizon: Later horizon of `c1` or `c2`
Acquisition: Immediately acquire `c1` and `c2`

`or :: Contract → Contract → Contract`
Acquiring ``c1 `or` c2`` means the holder must immediately choose to acquire either `c1` or `c2`. They cannot acquire both or neither. If one contract has expired, it cannot be chosen.
Horizon: Later horizon of `c1` or `c2`
Acquisition: Immediately acquire `c1` or `c2`

`then :: Contract → Contract → Contract`
Acquiring ``c1 `then` c2`` means acquiring `c1` if it has not expired. If `c1` has expired, you acquired `c2`. If `c2` has also expired, the contract has expired.
Horizon: Later horizon of `c1` or `c2`
Acquisition: Immediately acquire `c1` or `c2` as described above

`scale :: Obs Double → Contract → Contract`
Acquiring `scale o c` means immediately acquiring `c` with all obligations of `c` multiplied by the value of `o` at the moment of acquisition.
Horizon: Horizon of `c`
Acquisition: Immediately acquired scaled `c`

`truncate :: Date → Contract → Contract`
Acquiring `truncate t c` means immediately acquiring `c` but with its horizon trimmed to time `t`. If the current time is after `t`, the contract cannot be acquired. Note that the rights and obligations of `c` may extend *beyond* the period during which it can be acquired. The horizon of the underlying contract is not itself mutated—it is *wrapped* in a contract with a shorter horizon.
Horizon: `t`
Acquisition: Immediately acquire `c` with a horizon of `t`

`get :: Contract → Contract`
If you acquire `get c` you must acquire `c` exactly at its horizon.
Horizon: Horizon of `c`
Acquisition: Wait until horizon of `c` to acquire `c`

`anytime :: Contract → Contract`
Acquire `anytime c` entails acquiring `c`, but at a time of the holder's choosing between the acquisition of `anytime c` and the horizon of `c`.
Horizon: Horizon of `c`
Acquisition: Acquire `c` at some time between now and horizon

Figure 2.1: The various combinators proposed by the original *Composing contracts* paper.

`cond :: Obs Bool → contract → Contract → Contract` Acquiring `cond o c1 c2` means acquiring `c1` if `o` is true at acquisition, else `c2`.	`anytime :: Obs Bool → Contract → Contract` Acquiring `anytime o c` permits to the holder to acquire `c` at a moment of their choice while `o` is true.
`when :: Obs Bool → Contract → Contract` Acquiring `when o c` means acquiring `c` immediately at the next instance when `o` becomes true.	`until :: Obs Bool → Contract → Contract` Acquiring `until o c` means acquiring `c` immediately. However, the contract (and any of its children) must be abandoned when `o` becomes true.

Figure 2.2: The new combinators introduced in *How to write a financial contract* [17]

By eliminating the more abstract `get` and `truncate` combinators, the zero-coupon bond becomes significantly easier to read. This extends to the European and American options:

```
european :: Date → Contract → Contract
european t u = when (at t) (u `or` zero)

american :: (Date, Date) → Contract → Contract
american (t1,t2) u = anytime (between t1 t2) u
```

While the contracts described here are fairly simple, it is evident that the *Composing contracts* primitives can be easily combined to construct quite complex contracts.

2.3 Smart contracts

Another key technology for this project is the 'smart contract'. At the most general level, smart contracts are some kind of computer-enforced system for enforcing, executing or negotiating contracts. The term was initially coined by Nick Szabo [18], but it is only in the last few years that blockchain technology has started to make real-life deployment of smart contracts feasible.

Ideally, smart contracts are self-enforcing, but there remain a number of open questions around how this can be achieved. However, blockchain technology does provide a number of useful properties for a smart contract platform. Smart contracts can now be made immutable and autonomous, and can be deployed to a decentralised network which does not require any trusted intermediary or escrow.

2.3.1 Origins of blockchain

The second core technology for this project is the Ethereum blockchain. In general, blockchains are a family of technologies for creating distributed, append-only databases. Initial work in this area took place in the early 1990s, when Haber and Stornetta described a scheme, using cryptographic hashing and signatures, for reliably timestamping documents [19]. Later, they improved their timestamping system with the addition of Merkle Trees [20].

In 1993, Dwork and Naor suggested that spam could be combatted by requiring email senders to compute a moderately difficult (but easy to verify) problem [21]. This would allow individuals to send email unimpeded, but make it impractical to send bulk messages. This concept—that of proving that you have performed a certain amount of computation—is known as 'proof of work', but it was a few years until Jakobsson and Juels formalised the idea [22].

Over the following years, there were a number of attempts to design a usable cryptocurrency. Wei Dai's *b-money* [23] protocol was one such attempt, but it relied on semi-trusted central servers and was never adopted. In 2008, Nick Szabo published a blog post describing a theoretical *bit gold*. This concept combined the idea of proof-of-work with an 'unforgeable chain of title' similar to a blockchain. Around the same time, the Bitcoin whitepaper was pseudonymously published by 'Satoshi Nakamoto' [24]. This document outlined the design of Bitcoin, the first—and still the most valuable—cryptocurrency. The Bitcoin network itself was launched in January 2009. After a few years, the concept of a cryptocurrency—the application—started to become distinct from that of a blockchain—the implementation.

To show how blockchains work, I will briefly describe Bitcoin's implementation.

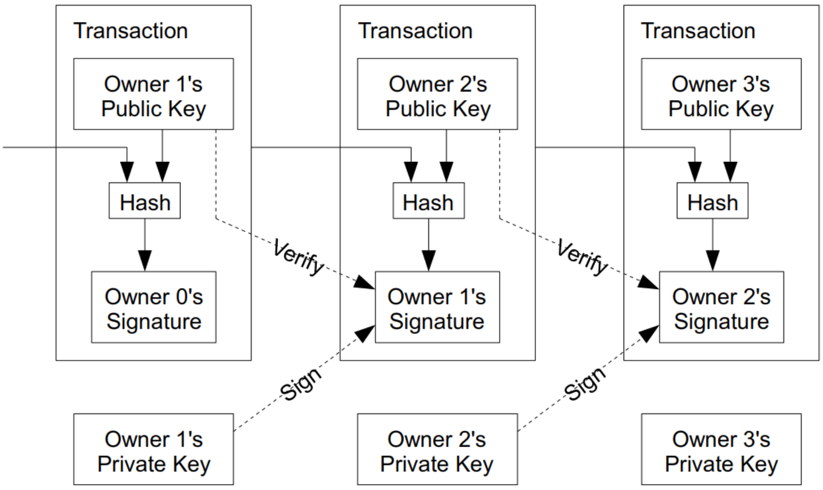

Figure 2.3: A series of signed transactions (source: Bitcoin whitepaper [24])

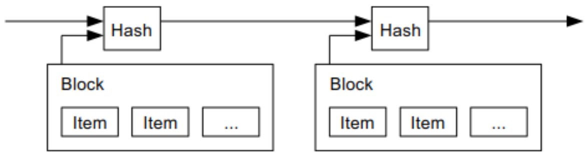

Figure 2.4: A series of blocks timestamped by a timestamp server (source: Bitcoin whitepaper [24])

Transactions

First, we need a way to prove that the owner of some electronic coin has consented to transfer it to another party. This is quite simple to implement using public-key cryptography. As the current owner of the coin, we hash the tuple of *(payee's public key, previous transaction)* and sign the hash with our private key. The payee can prove their ownership of the public key and thus that they are the intended payee. This is shown in Figure 2.3.

However, this scheme does not protect against a so-called 'double-spend' attack. The owner of a coin could sign and publish multiple transactions transferring the same coin to different payees. Without a central authority, there is no easy way to form a consensus on which is the 'correct' transaction.

Proof-of-work

For consensus, Bitcoin adopts a simple protocol: transactions must be announced publicly; everyone agrees on the order of the transactions and, in the case of double-spend, the (agreed) first transaction is the valid one. A payee must be able to ascertain that the majority of nodes in the network recognise the transaction.

In a centralised system, we could solve this problem with a 'timestamp server': a trusted system which timestamps a block of data and publishes its hash. This is shown in Figure 2.4. To implement a decentralised currency we must find a similar system with no central server.

For this, we can use a proof-of-work based system. In Bitcoin, participation in the proof-of-work challenge is known as *mining*. Each miner chooses a potential new *block* of transactions by taking the hash of the previous block, a group of non-conflicting transactions and a nonce. The miner then computes the SHA-256 hash of the block. To be published and recognised by the rest of the network, the hash must begin with a certain number of zero bits. If the hash does not meet this criterion, the nonce is incremented until a compliant hash is found. The difficulty of finding a hash with n zero bits is 2^n, but the difficulty of verifying a (finite) block is constant.

Repeating this process produces an ordered 'chain' of blocks, each containing new transactions, hence the term 'blockchain'. An honest Bitcoin client must respect the longest valid chain that it knows about. The blockchain structure is illustrated in Figure 2.5.

As a result, it is near-impossible for an attacker to modify past blocks or insert invalid blocks. Modification of a previous transaction will change the hash of a block. The hash of each block is an input

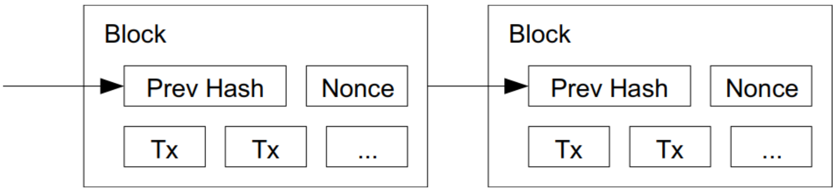

Figure 2.5: Bitcoin's basic blockchain structure (source: Bitcoin whitepaper [24])

to the hash of the next block. So, to modify a past transaction, the attacker must recompute all blocks since that transaction and 'overtake' the current longest chain. Statistically, this requires the attacker to control the majority of the network's computing power, and so it is known as a 51% attack[3]. Thus it is very expensive to attempt to manipulate old transactions—and the attacker could instead use that computing power as an honest miner. There is also a *cryptoeconomic* incentive to maintain the network's integrity as the value of the attacker's personal Bitcoin depends on this.

There are a few other properties of the blockchain that are important to understand. First, the successful miner for each block may award themselves a fixed quantity of coins. In addition, nodes that wish to have their transactions included in a block may offer a transaction fee. These are awarded to the miner as an incentive to include the transaction in their block. In Bitcoin, the mining difficulty increases over time while the block reward decreases.

The protocol has some other implementation details, such as the use of Merkle Trees [20] to reduce disk space requirements and the ability to have multiple inputs and outputs from a transaction, but these are not too important here.

2.3.2 What does Ethereum add?

Ethereum is a distributed computing platform that uses blockchain technology to maintain consensus across all its peers. In particular, Ethereum allows its users to write Turing-complete programs ('smart contracts') and deploy them to the blockchain in such a way that the state of the program is agreed on by every participant in the Ethereum network.

The state of the Ethereum network is a set of accounts, where each account has four properties [26]:

- A *balance*, measured in 'Ether' (the currency of Ethereum)
- A *nonce*, which counts transactions (to prevent replay attacks)
- *Storage*, empty by default
- *Contract code*, optional

There are two types of account. *External accounts* represent individuals (though one individual may own many accounts). The account owner has the private key for the account and may use it to perform transactions on the network. *Contract accounts* are controlled by their contract code. Any account can initiate a transaction[4] with the contract account, at which point the contract code is executed to determine the response. Some transactions may cause the contract's internal state to be mutated; others may cause the contract to initiate a transaction with another account—and so on.

It is important to understand that 'smart contracts' do not work like a traditional contract. They are not some kind of legal code that users must comply with. Instead, they act like autonomous entities on the Ethereum network, updating their internal state machine according to a well-defined set of rules.

2.3.3 Writing a smart contract

Having explored the high-level concepts of Ethereum, let's look at some very simple smart contracts. There are a number of ways to write an Ethereum smart contract, including:

Ethereum Virtual Machine (EVM) bytecode Eventually, all contracts are compiled to EVM bytecode.

[3] In fact, the GHash.io mining pool did once achieve >50% of the network's hashing power in 2014. [25]

[4] Creating a transaction is often also referred to as 'sending a message'. There is a subtle difference between the two, but it is not important here.

LLL A low-level Lisp-like language.

Serpent and Vyper Python-esque high-level languages.

Solidity High-level statically-typed language that shares some syntax with JavaScript. The most popular language by a large margin.

We will use Solidity, because it is easy to use and well-supported by a wide range of tools. Let's start with one of the simplest possible smart contracts, taken from the Solidity documentation [27]. It stores an integer value, which any account can update or retrieve by sending a message to the contract.

```
pragma solidity ^0.4.23;

contract SimpleStorage {
 uint storedData;

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

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

A `contract` is much like a class in an object oriented language. It can be instantiated and deployed to its own account on the Ethereum network. This particular contract has one state variable `storedData` and two methods, `set` and `get`. In order to change the stored value, any account can send a message calling the `set` method. In order to retrieve the value, they can send a message calling the `get` method.

By default, the visibility of `storedData` is `internal`. This means that it can only be retrieved from inside the contract itself. However, this does *not* make the data private. In fact, every Ethereum node has a full copy of the contract's state. `internal` visibility simply stops the Solidity compiler from generating the getter that would allow other contracts to easily access the value.

This simple contract highlights almost everything necessary to understand how smart contracts work in Ethereum. More complex functionality is achieved through language features analogous to those in most other modern programming languages. We can see a few more of Solidity's features by implementing an auction contract, again used as an example in the official documentation [28]. This is an open auction which holds the current highest bid in escrow until the auction ends. After the end of the auction, the seller can withdraw the escrowed bid.

First, we tell the compiler the minimum version with which this contract is compatible and declare the contract class.

```
pragma solidity ^0.4.23;

contract SimpleAuction {
```

Our action has a number of state variables. The `beneficiary` is the network address[5] of the account that will receive the money from the auction. The `auctionEnd` is an unsigned integer representing the Unix timestamp at which the auction should end.

`highestBidder` and `highestBid` are quite self explanatory, but `pendingReturns` uses an unfamiliar construct. A `mapping(address => uint)` is effectively a hash map: it maps every `address` value to a `uint`, with the `uint` default initialised to zero. The `uint` represents the sum of an account's superseded bids—bids which have been held in escrow but can now be returned to the account.

```
 address public beneficiary;
 uint public auctionEnd;

 address public highestBidder;
 uint public highestBid;

 mapping(address => uint) pendingReturns;

 bool ended;
```

Next, we define a couple of events that can be emitted by the contract. Ethereum is designed such that it is easy for clients to subscribe to events (for example, in order to update a UI or trigger some action).

[5] The last 20 bytes of the account's public key.

```
event HighestBidIncreased(address bidder, uint amount);
event AuctionEnded(address winner, uint amount);
```

Finally, we define various functions for the auction contract. The constructor takes the length (in seconds) of the auction and the beneficiary address and sets up the initial state of the contract.

```
constructor(uint _biddingTime, address _beneficiary) public {
  beneficiary = _beneficiary;
  auctionEnd = now + _biddingTime;
}
```

The `bid` function does not take any parameters, but is marked `payable`. Any transaction that calls this function can include some amount of Ether. The amount of Ether included *is* the bid, and is held in escrow by the contract. The function is also marked `public`. This means that any account can call the function. Functions can be made `private` or `internal`, which prevents them from being called externally.

```
function bid() public payable {
  require(now <= auctionEnd, "Auction already ended.");
  require(msg.value > highestBid, "There is already a higher bid.");

  if (highestBid != 0) {
    pendingReturns[highestBidder] += highestBid;
  }

  highestBidder = msg.sender;
  higestBid = msg.value;
  emit HighestBidIncreased(msg.sender, msg.value);
}
```

Notice how this function uses `require`. This is a bit like assertion in many other programming languages—the supplied expression must evaluate to true otherwise the function execution will be rolled back. Function execution in Solidity is transactional, so exceptions cause any state changes to be reverted. If the current time is beyond the end of the auction or if the ether value sent with the transaction is not higher than the current highest bid, calling the function will have no effect.

If the new bid is higher, however, the current highest bid is moved to the mapping of pending returns and the contract updates its internal state to reflect the new highest bid. It also emits a `HighestBidIncreased` event, which may be useful for other auction participants.

```
function withdraw() public returns (bool) {
  uint amount = pendingReturns[msg.sender];
  if (amount < 0) {
    pendingReturns[msg.sender] = 0;

    if (!msg.sender.send(amount)) {
      pendingReturns[msg.sender] = amount;
      return false;
    }
  }
  return true;
}
```

The `withdraw` function allows previous bidders to withdraw their superseded bids. Notice that the `pendingReturns` slot is set to zero before `send` is called, and is reset back to `amount` only if the sending fails. This is because accounts can have a custom function for receiving Ether, which could allow the receiver to mount a reentrancy attack by recursively calling `withdraw`.

Finally, we define a function for finalising the auction.

```
  function auctionEnd() public {
    require(now >= auctionEnd, "Auction not yet ended.");
    require(!ended, "auctionEnd has already been called.");

    ended = true;
    emit AuctionEnded(highestBidder, highestBid);

    beneficiary.transfer(highestBid);
  }
}
```

This auction example highlights the potential of smart contracts. The contract eliminates the risk of an untrustworthy auction house—instead, the behaviour of the contract is agreed by every Ethereum network participant. Once deployed, the contract is immutable. As a result, its behaviour is completely deterministic. This means that a bidder can be sure that her superseded bids will be returned. Equally, the beneficiary can be assured that the highest bid will be paid to them.

It is clear that similar contracts could readily applied to various problems, such as voting, various other types of auction, gambling/prediction markets, escrow and so on.

However, this contract also raises some questions. Why is the bid held in escrow? It is not possible for the contract to forcibly extract a payment post-hoc—perhaps the bidder's balance is zero at the end of the auction (it cannot be negative). In general, it is very difficult to enforce debt through a smart contract. In the real world, debt repayment is enforced through the legal system. Creditors use reputational systems such as credit scoring to avoid lending to parties who are unlikely to repay[6].

To avoid repayment, unscrupulous actors sometimes take out loans through limited-liability shell companies with few assets. When creditors try to retrieve their debts, the company cannot pay and is liquidated. However, there are a number of safeguards: in particular, company directors can be held personally liable by a court if their actions appear to be fraudulent. Individuals can also be disqualified as company directors, so repeating this attack requires a constant supply of new people.

Unlike humans, it is very fast and cheap to generate new Ethereum accounts. In peer-to-peer systems, this is known as a Sybil attack, and the abundance of new identities makes it harder to design robust reputation systems. Finding solutions to this problem is an area of ongoing research [29]–[32].

Similarly, notice that the contract cannot enforce the delivery of the goods purchased through the auction. It is difficult to find ways to connect real world objects to a blockchain, but there are some easier cases. A smart door or rental vehicle could verify payment on the blockchain and permit usage to anyone who could prove ownership of the relevant private key. This approach is being pursued by Slock.it GmbH [33]. It may be instead that blockchains become widely accepted as evidence of ownership transfer—this would make it much easier to settle some legal disputes.

2.3.4 Paying for transactions

As discussed above, Bitcoin allows users to include a fee to incentivise completion of their transactions. Ethereum has a similar mechanism for every transaction, including calls to smart contract methods. The Ethereum Yellow Paper [34] specifies the cost, in 'gas', for each EVM operation. For example, a *CALL* operation costs 700 gas, while a *sha3* operation costs 30 gas. The cost, in gas, of executing any smart contract method is deterministic: it is determined precisely by the EVM operations that will be executed. However, the user can choose how much to pay *per unit gas*. Thus the actual transaction fee is (gas used) $\times$ (gas price).

[6] This is termed 'mitigating counterparty risk'.

Chapter 3

Implementation

Now that we understand the composing contracts language, we can start building our own version. The first step is to find a way to embed this domain-specific language inside Haskell in a way that is modular, extensible and which makes it easy to write a compiler. We choose to create an embedded DSL (rather than a standalone language) because it allows us to reuse Haskell's advanced type system and flexible syntax. Second, we look at writing various interpreters for the language—including a compiler to Solidity. Finally, we develop a GUI client to help us deploy and manage our compiled contracts.

My final implementation of the language, compiler and client is called *Merchant*.

3.1 Embeddings

There are two key types of DSL implementation: shallow and deep. These terms were coined by Richard Boulton in 1992 to describe two approaches to embedding a hardware description language (HDL) in higher-order logic [35], [36]. A *shallow embedding* was characterised by implementing the semantic operations of the HDL directly in higher-order logic. A *deep embedding* involved using terms in higher-order logic to represent the abstract syntax tree (AST) of the HDL. A semantic interpretation of the AST could then also be implemented in the higher-order logic. In this chapter, we will explore several embeddings and their relative tradeoffs.

In the Haskell community, it is understood that a shallow embedding implies the use of functions[1] to represent operations in the DSL. The composing contracts language described by Peyton Jones and Eber is an excellent example of such a language. Recall, for example, that the `truncate` combinator is defined as:

```haskell
truncate :: Date → Contract → Contract
```

It is interesting to note that the *type* of the function is specified, but the *implementation* is not. This raises a question: what *is* the correct implementation of this function? It depends on how we want to interpret the contract definition. This is the same as asking: what is the definition of the `Contract` type?

3.2 A shallow embedding

If we simply want to print a representation of the contract, it might be obvious to choose the following representation:

```haskell
type Contract = String

truncate :: Date → Contract → Contract
truncate d c = "Truncate(" ++ show d ++ "," ++ c ++ ")"
```

Here, we use a *type synonym* to indicate to the compiler that `Contract` and `String` are equivalent. However, what if we then want to implement a different interpretation? Perhaps we are interested in finding the number of primitives used in a contract:

```haskell
type Contract = Int

truncate :: Date → Contract → Contract
```

[1]Often the term *combinator* is used.

```haskell
truncate _ c = 1 + c

and :: Contract → Contract → Contract
and c1 c2 = 1 + c1 + c2
```

This presents a clear problem: we cannot have two different definitions of the same function. How could we overcome this? Perhaps the answer is to return all the interpretations we need from the same function:

```haskell
type Contract = (String, Int)

truncate :: Date → Contract → Contract
truncate d c = ("Truncate(" ++ show d ++ "," ++ c ++ ")", 1 + c)
```

But this is an unwieldy solution. Imagine trying to implement an interpreter that had a several independently toggleable optimisations. You would be forced to maintain a huge tuple of largely-identical interpretations for every possible permutation of options.

3.3 A deep embedding

Deep embedding is a solution to this problem. In a deep embedding, the DSL is represented as a Haskell data structure which can then be interpreted by other functions.

```haskell
data Contract
  = Zero
  | One Currency
  | Give Contract
  | And Contract Contract
  | Or Contract Contract
  | Truncate Date Contract
  | Then Contract Contract
  | Scale (Obs Double) Contract
  | Get Contract
  | Anytime Contract
  deriving (Eq, Show)
```

This data declaration says that a value of type `Contract` can be constructed by calling `Zero`, `One k` etc. With this defined, we can now rewrite our previous interpreters:

```haskell
render :: Contract → String
render Zero = "Zero"
render (One currency) = "One(" ++ show currency ++ ")"
render (Give c) = "Give(" ++ render c ++ ")"
render (And c1 c2) = "And(" ++ render c1 ++ "," ++ render c2 ++ ")"
render [ ... ]

count :: Contract → Int
count Zero = 1
count (One _) = 1
count (Give c) = 1 + (count c)
count (And c1 c2) = 1 + (count c1) + (count c2)
count [ ... ]
```

3.3.1 Netrium

This approach is used by Netrium [37], which implements a language strongly inspired by composing contracts. It has a number of advantages over a shallow embedding. Now we can easily define multiple interpreters for our DSL while retaining a very simple implementation. In addition, it becomes possible to serialise the DSL at runtime—useful if we want to save a contract to disk or send it over a network.

However, the deep embedding does change the appearance of our language somewhat. Assuming that observables are also represented by a deep embedding, our original zero-coupon bond definition might change as so:

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

zcb' :: Date → Double → Currency → Contract
zcb' t x k = Scale (Const x) (Get (Truncate t (One k)))
```

To retain the original syntax, we simply need to define some constructor functions for our `Contract` and `Obs` data types. Here are some examples:

```haskell
scaleK x c = Scale (Const x) c
get c = Get c
truncate t c = Truncate t c
one k = One k
```

Notice that writing the constructors is a very mechanical process—this could be done automatically through metaprogramming.

3.3.2 Findel

Biryukov et al. [38] have proposed an alternative method for implementing the composing contracts language in Ethereum. They use a deep embedding in the Solidity language, which they call Findel. This makes it possible to write contracts as Solidity expressions. For example, a zero-coupon bond becomes:

```
function Zcb(uint _exactTime, int _scaleCoeff, Currency _curr) returns(bytes32) {
return At(_exactTime, Scale(_scaleCoeff, One(_curr));
}
```

The authors highlight some limitations in the Ethereum platform. For example, there is not a precise clock available to smart contracts, only the timestamp of the latest block. While blocks are mined every 15 seconds, miners have the ability to influence timestamps somewhat. This could have serious implications for the fairness of contract execution.

In addition, they note that Solidity is a fundamentally imperative language with a fairly weak type system. While this does have some impact on our implementation, our implementation will mitigate it somewhat by transpiling contracts from Haskell. In addition, we will take a rather different approach to contract representation in Solidity that makes it possible to include more complex contract primitives.

3.4 Extending the language

Our deep embedding seems like a good solution to the problem of representing a DSL. However, it begins to present some problems if we wish to extend the language. In fact, composing contracts is a great case study for exactly this problem. The original and updated paper propose two different sets of primitives. Some primitives, like `zero` and `or`, are present in both versions. Some, like `get` and `truncate`, appear only in the original version. Others, like `when` and `until`, are introduced by the updated version. In addition, there are some subtle differences in semantics: the updated paper eliminates the concept of the contract horizon.

Now, think about how we could extend the original deep embedding. The simple approach is to just add the new primitives:

```haskell
data Contract
  = Zero
  | [...]
  | Anytime Contract
  | Cond (Obs Bool) Contract Contract
  | When (Obs Bool) Contract
  | AnytimeO (Obs Bool) Contract
  | Until (Obs Bool) Contract
deriving (Eq, Show)
```

There are some problems here that are immediately clear. First, our interpreters (`render` and `count`) are now broken. Previously, `render` was a total function—that is, it was defined for any possible (finite) `Contract`. After adding new constructors to the `Contract` type, it has become partial function. We must resolve this for every function that takes `Contracts` as input by adding a code path for `Cond`, `When` and so on. If we do not do this, our interpreters may throw type errors at runtime.

However, there is a second problem. The updated language is not a superset of the original language—some primitives have been removed. This means that our `Contract` data type now represents a hybrid language that was never proposed by Peyton Jones and Eber. Perhaps we should separate the two languages into a `Contract` and `Contract2` datatype:

```haskell
data Contract
  = Zero
  | One Currency
  | And Contract Contract
  | Or Contract Contract
  | Truncate Date Contract
  | Then Contract Contract
```

```
  | Scale (Obs Double) Contract
  [ ... ]

data Contract2
  = Zero
  | One Currency
  | And Contract Contract
  | Or Contract Contract
  | Scale (Obs Double) Contract
  | Cond (Obs Bool) Contract Contract
  [ ... ]
```

This solves the second problem: both `Contract` and `Contract2` now map precisely to the languages proposed in one of the papers.

3.4.1 The Expression Problem

However, the first problem remains. In fact, it is now even worse! We must implement a `render :: Contract → String` and `render2 :: Contract2 → String`, and are forced to duplicate the logic for handling identical primitives that appear in both data types.

This issue is a specific instance of what is known as the 'expression problem', a term coined by Philip Wadler in 1998 [39]. The expression problem concerns the degree to which 'your application can be structured in such a way that both the data model and the set of virtual operations over it can be extended without the need to modify existing code, without the need for code repetition and without runtime type errors' [40].

For example, imagine that we are trying to represent the stock of a local cheese emporium. In Haskell, we might represent the different kinds of cheeses like so:

```
data Cheese
  = RedLeicester
  | Caerphilly
  | Camembert
```

In order to determine the current status of each cheese, we can implement a function like so:

```
status :: Cheese → String
status cheese = case cheese of
  RedLeicester → "fresh out"
  Caerphilly → "been on order for two weeks"
  Camembert → "a bit runny"
```

Now let's consider an equivalent implementation in a fictional object-oriented language. We will use a `Cheese` interface implemented by a number of concrete cheese classes.

```
interface Cheese:
  status() :: String

class RedLeicester:
  status() = "fresh out"

class Caerphilly:
  status() = "been on order for two weeks"

class Camembert:
  status() = "a bit runny"
```

Until now, both implementations seem roughly equivalent. However, let's look at how they deal with two different types of extension. First, consider adding a new cheese (call it `CzechoslovakianSheepsMilkCheese`). The object-oriented case is easy, since we simply add a new class. The functional case, however, is more difficult: we must handle the new case in any function that takes a `Cheese` value. If we do not, our code will contain partial functions—which may crash at runtime.

Second, we can consider the impact of adding a new method. Let's call it `origin`, and have it return the location in which the cheese originated. This is easy for the functional case: we simply write a new function `origin :: Cheese → String`. However, it is now more difficult for the object-oriented implementation. Once we add the new method `origin() :: String` to the `Cheese` interface, we must implement it for every class—or our code will not compile.

An ideal solution to the expression problem would somehow give us the 'best of both worlds'. Many have been proposed over the years, but we will focus on one that is popular for DSLs implemented in

Haskell. We will reimplement our DSL data type in a free monadic style and use the Data Types á la Carte approach [11] to enable extensibility and modularity in our language.

3.4.2 Modularising the language

To solve the expression problem, we must ensure that we do not fix ourselves to a concrete set of data constructors. This means that we need to somehow create a constructor that is parametrised by the type we actually want to construct. Here I will explain the Data Types á la Carte approach.

Consider, for example, that we are trying to represent just a couple of the composing contracts primitives: `One` and `And`. We'll start by defining a couple of data types to represent these two primitives:

```
data One k = Val Currency deriving Functor
data And k = And k k deriving Functor
```

At the moment, `And` is non-recursive. However, you might notice that the constructor takes two values of some type `k`. We could construct a value such as `And (And ()) (And ())` to get two layers of addition. This would have type `And (And ())`. Notice also that we allow the compiler to make the datatypes into `Functors`. This allows us to use `fmap` to apply a function to the `k` value inside the datatype.

Clearly, this approach will become unwieldy rather quickly, as the type of the expression is still dependent on the structure of each individual expression. To begin solving this, we introduce the `Fix` data type.

```
Fix :: (* → *) → *
data Fix f = Fix (f (Fix f))
```

`Fix` is parametrised by `f`, which has kind `* → *`—this means that `f` takes a type and returns another type. Notice that our `One` and `And` definitions are also kind `* → *`.

If we apply `Fix` to `One` or `And`, we effectively obtain an infinitely telescoping type, but without changing the type signature of the expression.

```
Fix One ≃ Fix (One (Fix (One (Fix ( … )))) :: Fix One
Fix And ≃ Fix (And (Fix (And (Fix ( … )))) (Fix (And (Fix ( … ))))) :: Fix And
```

Let's give those `Fix`ed types some friendlier aliases, and see if we can construct a value of each type.

```
type OneExpr = Fix One
type AndExpr = Fix And

let e1 = Fix (One USD) :: OneExpr
let e2 = Fix (And (Fix (And ( … ))) (Fix (And ( … )))) :: AndExpr
```

There are a couple of problems here. First, we can't combine `And` and `One` in a single expression. Second, we cannot construct a finite expression using `And` at all, because it must be an infinite tree of `And`s. It turns out that solving the first problem will solve the second: if we can use both `One` and `And` in our expression, we can use `One`s terminate the tree of `And`s.

To do this, we will define a data type that represents the *coproduct* of two type constructors.[2]

```
data (f :+: g) e = L (f e) | R (g e) deriving Functor
```

Now we can actually construct an expression of both `One`s and `And`s.

```
contractExample :: Fix (One :+: And)
contractExample = Fix (R (And (Fix (L (One USD))) (Fix (L (One GBP)))))
```

Now we can think about how to write an interpreter for our expressions. We have let the compiler derive the `Functor` instance for `One`, `And` and `f :+: g`. Given an *algebra* of type `Functor f ⇒ f a → a`, we can now fold any contract expression to get a single value of type `a`. The `Functor f ⇒` here simply means that the type `f` must always be a `Functor`.

```
handle :: Functor f ⇒ (f a → a) → Fix f → a
handle alg (Fix t) = alg (fmap (handle alg) t)
```

Notice that the algebra receives input values that are *already* of the output type. We are evaluating our expression from the bottom up without the use of general recursion.

Now if we wish to write some interpreter for the contract expression (in this case, rendering to a string), it is a simple matter of defining the algebras for each type.

[2] Note that we could write something like `data Coproduct f g e = L (f e) | R (g e)`. We are using the GHC `TypeOperators` extension to get `:+:`.

```haskell
class Functor f => Eval f where
  evalAlg :: f String -> String

instance Eval One where
  evalAlg (One currency) = "One(" ++ show currency ++ ")"

instance Eval Add where
  evalAlg (Add x y) = "Add(" ++ x ++ "," ++ y ++ ")"

instance (Eval f, Eval g) => Eval (f :+: g) where
  evalAlg (L x) = evalAlg x
  evalAlg (R y) = evalAlg y

eval :: Eval f => Fix f -> String
eval expr = handle evalAlg expr
```

Here we have defined a *typeclass* called `Eval`. A typeclass simply defines some functions involving a to-be-decided type (in this case, `f`). We use the `instance` declarations to implement this typeclass for the types `One`, `Add` and `f :+: g`. `Functor`, which we encountered earlier, is also a typeclass.

3.4.3 Better modularisation

There are a few remaining questions. Is it reasonable to expect a developer to construct expressions in the manner of `contractExample`, or could we automate this?[3] Also, how do we deal with more than two types at a time?

The answers to these questions turn out to be closely linked. First, let's think about how we could support more than two types. The Data Types à la Carte solution is to use a tree of coproducts like `f :+: (g :+: (h :+: i))`. For ease of implementation, we will only support right-recursive trees.

It would be useful to define a typeclass which witnesses the subtype-supertype relationship between two types. In order to do this, we must be able to define an *injection*—a way of converting any value inhabiting the subtype into a corresponding value inhabiting the supertype.

```haskell
class (Functor sub, Functor sup) => sub :<: sup where
  inj :: sub a -> sup a

-- a value in f is trivially convertible to one in f
instance Functor f => f :<: f where
  inj = id

-- a value in f is convertible to one in (f :+: g)
instance (Functor f, Functor g) => f :<: (f :+: g) where
  inj = L

-- a value in f is convertible to one in (h :+: g) if f is convertible to g
instance (Functor f, Functor g, Functor h, f :<: g) => f :<: (h :+: g) where
  inj = R . inj
```

With `inj` defined for all cases, we can use it to create 'smart' constructors for `One` and `And`.

```haskell
inject :: (g :<: f) => g (Fix f) -> Fix f
inject = Fix . inj

one' :: (One :<: f) => Currency -> Fix f
one' x = inject (One x)

and' :: (And :<: f) => Fix f -> Fix f -> Fix f
and' x y = inject (And x y)
```

Once we have wrapped the value with `One` or sub-expressions with `And`, we can see that `inject` does most of the heavy lifting.

Finally, we can illustrate how easy it is to now extend the expression language. Let's say we want to add `Scale`. Here's the code:

```haskell
data Scale k = Scale (Obs Int) k deriving Functor

instance Eval Scale where
  evalAlg (Scale o c) = "Scale(" ++ show o ++ "," ++ c ++ ")"

scale' :: (Scale :<: f) => Fix f -> Fix f -> Fix f
scale' o c = inject (Scale o c)
```

[3] No and yes, respectively.

Notice that it does not matter if we decide to delay implementing the `Eval Scale` instance. Evaluation of `Fix (One :+: And)` expressions will still work, and be completely typesafe. In the instance where we want to include `Scale` in an expression, we can simply switch to using the type `Fix (One :+: And :+: Scale)`. At this point, the compiler will force us to specify the `Eval` instance for `Scale` before we can call `eval` on it.

It is now possible to precisely and safely implement interpreters for different subsets of our language. In addition, adding a new interpreter is just a matter of defining and implementing another typeclass like `Eval`.

3.5 Freedom

At the moment, our `Fix`-based expressions do not allow us to easily express sequential logic with side effects. In fact, since this is not a feature of the original composing contracts language, we could stop here. However, we will continue on for two reasons. First, free monads make it easy to do so; second, it might allow us to add some useful features to the language.

In Haskell, monads are commonly used to structure sequential and effectful logic. Luckily, having defined our language primitives in terms of functors, we can get monads 'for free'. First, let's explore what 'free' means.

Recall that an *algebraic structure* on some 'carrier' set A is a collection of finitary operations[4] on elements in the set [41]. Specific algebraic structures include *groups* and *rings*.

Another type of algebraic structure common in functional programming is the *monoid*. Let a monoid on the carrier set A be a tuple $(A, \circ, e)$. The binary operation $\circ$ has type $S \times S \to S$. The element $e \in A$ is an identity element. The monoid must obey two laws:

$$\forall a, b, c \quad (a \circ b) \circ c = a \circ (b \circ c) \tag{3.1}$$
$$\forall a \quad a \circ e = e \circ a = a \tag{3.2}$$

In Haskell, monoids are defined by the `Monoid` typeclass

```haskell
class Monoid m where
  mempty :: m
  mappend :: m → m → m
```

where `m` corresponds to A, `mempty` to e and `mappend` to $\circ$. A simple example of a monoid is $(\mathrm{Z}^{+}, \times, 1)$. The monoid axioms hold: multiplication is commutative, and any positive integer multiplied by 1 remains the same.

We say that the monoid's properties ($\circ$ and e) add extra 'structure' to the set A. A free `something` allows us to add extra structure to a less structured object 'for free'. For example, given *any* Haskell type `t`, we can construct a monoid `[t]`:

```haskell
instance Monoid [t] where
  mempty = []
  mappend = (++)
```

In category theory, a free object is one where only the minimal properties required by the object hold. For example, the free monoid has only the properties in Equations 3.1 and 3.2. Though we have just implemented the free monoid in Haskell as a list, Mac Lane defines it more generally as a sequence: 'all the finite strings $x_1 x_2 \ldots x_n$ of elements x_i of the set X' [42].

This brings us on to the free monad. The free monad allows us to add extra structure to a functor, turning it into a monad [43]. In Haskell, we implement `Free` as follows:

```haskell
data Free f a = Pure a | Free (f (Free f a))

instance Functor f ⇒ Functor (Free f) where
  fmap f (Pure x) = Pure (f x)
  fmap f (Free t) = Free (fmap (fmap f) t)

instance Functor f ⇒ Applicative (Free f) where
  pure = return
  (<*>) = ap

instance Functor f ⇒ Monad (Free f) where
```

[4] Operations which take a finite number of inputs

```
  return x = Pure x
  (Pure x) >>= f = f x
  (Free t) >>= f = Free (fmap (>>= f) t)
```

Notice that, much like how the free monoid builds up a sequence of values, the free monad builds up a nested `Free` structure without performing any actual computation.

3.6 Going monadic

Now we can experiment with adding some effectful primitives to our language. To keep things simple, we'll just add the ability to store and retrieve an integer value.

```
data GetInt k = GetInt (Int -> k) deriving Functor
data SetInt k = SetInt k deriving Functor
```

We can add smart constructors for these, too. Notice that the smart constructors are a little different to those for `Fix`—they leave the continuation `Pure` so that it can be filled in by the monadic bind later on.

We also define a function `inject`, which combines the effects of `Free` and `inj`. Essentially, it uses `inj` to inject a value from some subtype `g` to a supertype `f`, then wraps this in the `Free` constructor.

```
inject :: (g :<: f) => g (Free f a) -> Free f a
inject = Free . inj

getIntM :: (GetInt :<: f) => Free f Int
getIntM i = inject (GetInt Pure)

setIntM :: (SetInt :<: f) => Int -> Free f ()
setIntM i = inject (SetInt i (Pure ()))

handle :: Functor f => (a -> b) -> (f b -> b) -> Free f a -> b
handle pure alg (Pure x) = pure x
handle pure alg (Free t) = alg (fmap (handle pure alg) t)
```

This allows us to use Haskell's do-notation to build up sequential programs. Assuming we have implemented monadic smart constructors for the other primitives, we could write something like this:

```
program :: Free (One :+: And :+: Scale :+: GetInt :+: SetInt) Int
program = do
  setIntM 5
  scaleBy <- getIntM
  scaleKM scaleBy
  oneM GBP
```

The do-notation syntax allows us to sequence a number of operations, as well as binding values that can then be used in the rest of the sequence (for example, `scaleBy <- getIntM`).

We can use do-notation to construct a contract even if there is no sequential or effectful logic. For example, here are two different ways to write an equivalent zero-coupon bond:

```
zcbOriginal :: Time -> Int -> Currency -> Contract
zcbOriginal t x k = scaleK' x (get' (truncate' t (one' k)))

zcbOriginalM :: Time -> Int -> Currency -> Contract
zcbOriginalM t x k = do
  scaleKM x
  getM
  truncateM t
  oneM k
```

Note that we can't yet handle two-argument primitives (e.g. *Or*) very cleanly in do-notation, but it should be possible to solve this with a more complex `Monad` instance.

3.7 Observables

Alongside the contract primitives themselves, the composing contracts language also makes heavy use of observable values to define the behaviour of contracts. While the embedding of the combinators is not explicitly specified in the original papers, the observables are clearly intended to be implemented as a shallow embedding. The authors go as far as to lift Haskell functions directly into the domain of the observables. For example, you could write:

Figure 3.1: An initial attempt at rendering a contract graph.

```haskell
let o :: Obs Int = lift2 (+) (konst 5) (konst 4)
```

This is not a representation that we can compile[5]—instead, a deep embedding is needed. I experimented with implementing a free monadic deep embedding similar to that used for the contract primitives. However, this approach currently has some intractable issues. It turns out to be very complex to define a free structure where we can combine `Obs a` values of different concrete types (e.g. `Obs Int`, `Obs Bool`). There is ongoing work [44] that may make this a viable approach in the future.

It would be undesirable to fall back to using concrete observable types like `ObsInt` or `ObsBool`. Thankfully, there is a middle ground which allows us to retain the `Obs a` interface in a somewhat constrained form. Using Haskell's GADT syntax, we can write the following:

```haskell
data Obs a where
  External :: String -> Obs a
  Constant :: a -> Obs a
  After :: Time -> Obs Bool
```

When the constructor produces an `Obs a`, type inference will allow the compiler to choose the correct concrete type in many circumstances. In order to simulate the behaviour of `lift`ed functions, we need to add new constructors. For example:

```haskell
  OAnd :: Obs Bool -> Obs Bool -> Obs Bool
  OGreaterThan :: Obs Int -> Obs Int -> Obs Bool
  OSubtract :: Obs Int -> Obs Int -> Obs Int
```

By way of comparison, `OAnd` is equivalent to `lift2 (&&)` in the original composing contracts language.

3.8 Writing interpreters

Now we can think about writing a more complicated interpreter. Let's consider rendering—instead of a textual representation—a graph of the contract. To add a new interpreter, we must define a function of type `Functor f => f a -> a`, where `a` is the type that our interpreter will eventually produce.

A good option for graph rendering in Haskell is Graphviz, a mature C library which has a Haskell binding of the same name. Graphviz uses a domain-specific language, DOT, to express the structure and organisation of graphs. The Haskell Graphviz library offers a monadic interface, `DotM n a`, to the DOT language. Each node in the graph has type `n`, and the value is used as a unique identifier when constructing the edges of the graph.

Our first attempt to write a graph renderer might look something like this:

```haskell
renderToGraphAlg :: Contract -> FilePath -> IO FilePath
renderToGraphAlg contract = runGraphviz dotGraph Png
  where dotGraph = digraph (Str "contract") (handle pure graphAlg contract)

class Functor f => GraphAlg f where
  graphAlg :: f (DotM String ()) -> DotM String ()

instance GraphAlg ContractF where
  graphAlg (One k) = do
    node "One" [A.textLabel (T.pack $ "One(" ++ show k ++ ")")]
  graphAlg (And c1 c2) = do
    node "And" [A.textLabel "And"]
    c1
    c2
  [...]
```

However, there are some problems with this implementation. If we render the contract `and' (one' USD) (one' USD)`, we get the graph shown in Figure 3.1. First, we have no way of creating the edges between nodes. Second, because the string 'One(USD)' is treated as if it is a unique identifier, that node only appears once in the graph—despite having two instances in the contract itself.

[5] Not without writing a new Haskell compiler, at least.

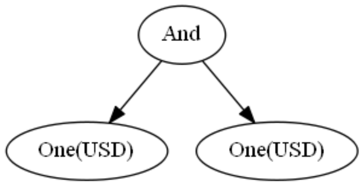

Figure 3.2: A contract graph render using the `State` monad.

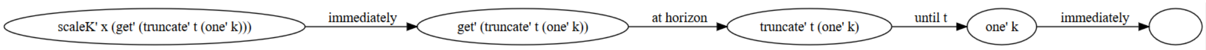

Figure 3.3: The progression of a nested zero-coupon bond contract.

To solve these problems we can wrap the `DotM` monad with the `StateT` monad transformer. Having the `State` monad available allows us to thread some persistent state through our entire interpreter. Due to the way monad transformers work, we can use `State` actions (e.g. `get`, which retrieves the current state) directly in do-notation. However, we must `lift` any `DotM` actions (e.g. `node`, which adds a node to the graph) into the `StateT` wrapper.

Using an incrementing integer as the state, we give each node a unique identifier. The $\longrightarrow$ operator allows us to create edges between nodes. The result, created with the following algebra, can be seen in figure 3.2.

```haskell
renderToGraphAlg :: Contract -> FilePath -> IO FilePath
renderToGraphAlg contract = runGraphviz dotGraph Png
  where graphState = handle pure graphAlg contract
  where dotGraph = digraph (Str "contract") (evalStateT graphState 0)

class Functor f => GraphAlg f where
  graphAlg :: f (StateT Int (DotM Int) ()) -> StateT Int (DotM Int) ()

-- Retrieve the current value of the unique ID counter from the state, then post-increment it.
increment :: StateT Int (DotM Int) Int
increment = do
  identifier <- get
  put (identifier + 1)
  return identifier

instance GraphAlg ContractF where
  graphAlg (One k) = do
    n <- increment
    lift $ node n [A.textLabel (T.pack ("One(" ++ show k ++ ")"))]
  graphAlg (And graph1 graph2) = do
    n <- increment
    graph1
    m <- get
    graph2
    lift $ node n [A.textLabel "And"]
    lift $ n --> (n + 1)
    lift $ n --> m
  [...]
```

In this code, you can see that the `graphAlg` sequences various actions: retrieving and incrementing the unique ID counter; including any subgraphs (for `And`); adding the current node and adding edges between the current node and the subgraphs.

3.9 Compiling to Solidity

There are a number of approaches that we could take to compile our contracts to Solidity. Our key design choice is to implement a nested series of Solidity `contract` objects, one for each combinator. Most combinators (e.g. `scale`) immediately disable themselves and deploy the next, as seen in Figure 3.3.

3.9.1 Marketplace

First, we will implement a *marketplace* contract to keep track of the balances of all contract parties. First, we declare a couple of data structures: `Commodity` represents the commodities that can be traded on this marketplace, and `ContractMetadata` represents the current state of a contract. The `counterparty` is the party that proposes (or sells) the contract, and the holder is the party that owns (or buys) the contract.

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

contract Marketplace {
  enum Commodity {USD, GBP}

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

  event Proposed(address contractAddress, address indexed to);
  event Signed(address contractAddress);
  event Delegated(address indexed from, address to);
```

We also store a few member variables. The `contracts_` mapping keeps track of all the contracts that have been proposed through the marketplace, while the `balances_` mapping keeps track of each individual user's balance in each commodity.

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

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

We now define a number of methods that allow users to create a new contract on the marketplace. The `propose` method allows a counterparty to propose a contract to another address. The `sign` method then allows the holder (to whom a contract is proposed) to agree to the contract, which immediately starts executing.

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

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

Last, we have a number of functions that can *only* be called by signed contracts. This guarantees that both parties have agreed to the actions of the contract calling these functions. Calling the `receive` causes the specified amount of some commodity to be transferred from the counterparty to the contract holder. The other two methods allow a contract to delegate authority to another contract. `delegate` simply adds a new entry to the `contracts_` mapping, whereas `give` reverses the counterparty and holder of a contract.

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

```

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

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

3.9.2 The base contract

In the `Marketplace` contract, we made some references to a `BaseContract`. This is an abstract base class for our compiled contracts that allows us to interact with them in a standard way.

Note a few important features of the `BaseContract`. It exposes the `proceed` method to continue the execution of the contract. This can be overridden with actual behaviour by each concrete implementation of `BaseContract`. In addition, `BaseContract` defines a `whenAlive` modifier that can be applied to any function. This will only permit a function to execute if the `alive_` member variable is true.[6]

```
contract BaseContract {
  event Killed();

  Marketplace public marketplace_;
  int public scale_;

  address public creator_;
  bool public alive_ = true;

  constructor(Marketplace marketplace, int scale) public {
    marketplace_ = marketplace;
    scale_ = scale;
    creator_ = msg.sender;
  }

  function proceed() public;

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

  function kill() internal whenAlive {
    alive_ = false;
    emit Killed();
  }

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

Note that `internal` functions can only be called by the smart contract itself: we don't want anyone to be able to `kill` a contract that's meant to execute. When defining the modifier, `_;` means 'now execute the rest of the function'.

3.9.3 Writing the interpreter

Now that we have some contract infrastructure in place, we can begin to write the interpreter itself. We will use a similar pattern to our graph interpreter, using the `State` monad to keep track of important information. For the compiler, though, we have more than one variable—so let's define a record:

[6] It would seem useful to be able to declare `proceed` with the `whenAlive` modifier. Unfortunately, however, modifiers are not inherited by overridden methods.

```haskell
data Solidity = Solidity {
 _source :: [Text], -- source code generated so far
 _counter :: Int, -- counter for unique contract naming
 _runtimeObservables :: [SType], -- observables that the contract needs to deploy at runtime
 _observableState :: ObsCompileState -- observables are pre-specified in the contract
}
makeLenses ''Solidity
```

Notice that we are using the *lens* library for this record—this allows us to update its fields more easily. Our algebra typeclass is defined as follows:

```haskell
class Functor f => SolidityAlg f where
 solidityAlg :: f (State Solidity (Text, Horizon)) -> State Solidity (Text, Horizon)
```

The return value is a `(Text, Horizon)` tuple representing the name and horizon of the current contract. The notion of a horizon appears only in the first composing contracts paper, so we must decide how to make the other primitives compatible with this concept. There are two main options:

Retrofit We add the notion of a horizon to the primitives introduced in the second paper. For example, *Get(When(someObs,Truncate(t,Zero)))* would be acquired at the earlier of *someObs* becoming true or time t being reached.

Transparency The primitives introduced in the second paper are 'transparent' to the horizon: they simply pass through the value of their children. For example, *get (when someObs (truncate t zero))* would be acquired at t regardless of when *someObs* becomes true.

In this implementation we choose the horizon-transparent option, as it is much easier to implement. It will also be useful to have a Haskell function to construct new concrete classes that derive from `BaseContract`. We use the *neat-interpolation* library to interpolate data into the contract skeleton (see the `[text| |]` blocks).

```haskell
makeClass :: Horizon -> T.Text -> T.Text -> T.Text -> T.Text -> T.Text
makeClass horizon className proceed members constructor =
 [text|
 contract ${className} is BaseContract {
  WrapperContract public wrapper_;
  bool public until_;
  BoolObservable private untilObs_;
  uint acquiredTimestamp_;
  ${members}

  constructor(Marketplace marketplace, int scale, WrapperContract wrapper, bool until, BoolObservable untilObs)
      public BaseContract(marketplace, scale) {
   wrapper_ = wrapper;
   until_ = until;
   untilObs_ = untilObs;
   acquiredTimestamp_ = block.timestamp;
   ${constructor}
  }

  function proceed() public whenAlive {
   if (until_) {
    bool untilFulfilled;
    (untilFulfilled,) = untilObs_.getFirstSince(marketplace_.isTrue, acquiredTimestamp_);
    if (untilFulfilled) {
     kill();
     return;
    }
   }
   ${horizonCheck}
   ${proceed}
  }
 }
 |]
 where
  horizonCheck = case horizon of
   Time t -> let t' = showt t in [text|
    if(now > ${t'}) {
     kill();
     return;
    }
   |]
   Infinite -> ""
```

Now we can easily implement some cases of the algebra. One of the simplest contracts is *One*. The `oneS` function generates the text of the contract itself, while the `solidityAlg` implementation updates the compiler state appropriately.

```
addClass cls sources = cls : sources

oneS :: Horizon → Currency → T.Text → T.Text
oneS horizon k n = makeClass horizon
  [text|One_${n}|]
  [text|
  marketplace_.receive(Marketplace.Commodity.${k'}, scale_);
  kill();|]
  ""
  ""
  where
    currency :: Currency → T.Text
    currency GBP = "GBP"
    currency USD = "USD"
    currency EUR = "EUR"
    k' = currency k

instance SolidityAlg ContractF where
  solidityAlg (One k) = do
    let horizon = Infinite
    counter %= (+1)
    n ← use counter
    source %= addClass (oneS horizon k (showt n))
    return ("One_" `T.append` showt n, horizon)
```

The majority of the rest of the implementation is mechanical, so we'll look at a couple of interesting cases here. First is *Or*, which requires creation of a runtime observable and has two subcontracts.

```
instance SolidityAlg ContractF where
  solidityAlg (Or c1 c2) = do
    (className1, horizon1) ← c1
    (className2, horizon2) ← c2
    let horizon = max horizon1 horizon2
    counter %= (+1)
    n ← use counter
    o ← showt . length <$> use runtimeObservables
    runtimeObservables %= (++ [SBool])
    let observableLiteral = [text|wrapper_.obs${o}_.getValue()|]
    source %= addClass (orS horizon className1 className2 observableLiteral (showt n))
    return ("Or_" `T.append` showt n, horizon)

orS :: Horizon → T.Text → T.Text → T.Text → T.Text → T.Text
orS horizon className1 className2 obs n = makeClass horizon
  [text|Or_${n}|]
  [text|
  if (${obs}) {
    ${className2} next2 = new ${className2}(marketplace_, scale_, wrapper_, false, BoolObservable(0));
    marketplace_.delegate(next2);
    next2.proceed();
  } else {
    ${className1} next1 = new ${className1}(marketplace_, scale_, wrapper_, false, BoolObservable(0));
    marketplace_.delegate(next1);
    next1.proceed();
  }
  kill();
  |]
  ""
  ""
```

Thanks to the way our interpreters are constructed, *Or* is quite simple to implement. First, we get the results of the two subcontracts `c1` and `c2`. The *Or* horizon is the later horizon of these two contracts. We increment the contract counter and also retrieve the count of `runtimeObservables` stored so far. We add another boolean observable to this list, then finally generate the contract source code.

As a result of some limitations in Ethereum, *When* is one of the most complex primitives to implement. Ethereum does not allow smart contracts to subscribe to other events in the blockchain: code execution must be initiated by an external account. This makes it difficult to implement *When*, which is meant to proceed immediately upon a boolean observable becoming true.

To solve this problem, we choose to adopt a slightly different execution strategy. We will permit either party to proceed contract execution at—and only at—the point when the boolean observable becomes true for the first time. If `proceed` is not called at this time, then the contract is considered *void by mutual agreement* and can never continue.

```
whenS :: Horizon -> T.Text -> Int -> T.Text -> T.Text
whenS horizon classId obsIndex n = makeClass horizon
  [text|When_${n}|]
  [text|
  bool fulfilled;
  uint when;
  (fulfilled, when) = obs_.getFirstSince(this.isTrue, acquiredTimestamp_);
  if (fulfilled) {
  if (when <= now && now <= (when + ${timeDelta})) {
      if (msg.sender == getHolder() || msg.sender == getCounterparty() || msg.sender == getCreator()) {
        BaseContract next = wrapper_.deploy(${classId}, marketplace_, scale_, wrapper_, false, BoolObservable(0));
        marketplace_.delegate(next);
        next.proceed();
        kill(BaseContract.KillReason.EXECUTED);
      }
    } else if ((when + ${timeDelta}) < now) {
      kill(BaseContract.KillReason.FAILED);
    }
  }
  |]
  [text|
  BoolObservable private obs_;

  function isTrue(bool input) external pure returns(bool) {
    return input;
  }
  |]
  [text|
  obs_ = wrapper_.deployBoolObservable(${obsIndex'});
  |]
  where obsIndex' = showt obsIndex

instance SolidityAlg ExtendedF where
  solidityAlg (When obs c1 c2) = do
    (className1, horizon1) <- c1
    (className2, horizon2) <- c2
    let horizon = max horizon1 horizon2
    obsConstructor <- zoom observableState (obsSolidityAlg obs)
    counter += 1
    n <- use counter
    source %= addClass (condS horizon className1 className2 obsConstructor (showt n))
    return ("When_" `T.append` showt n, horizon)
```

When executed, the `When` implementation essentially checks that now is the first time (within some delta) that the observable has been true. If it is, the underlying contract is instantiated. If not, nothing happens—unless the condition was met in the past, in which case the contract is killed.

3.9.4 Compiling observables

In our definition of `solidityAlg` for *When* we use `obsSolidityAlg` to update the `observableState`. This has type `ObsCompileState`, a record which contains the source code for each observable and a counter for creating unique identifiers.

```
data ObsCompileState = ObsCompileState {
  _obsSource :: [T.Text],
  _obsCounter :: Int
}
makeLenses ''ObsCompileState
```

We define a typeclass to be implemented for any `Obs a` that can be compiled to Solidity:

```
class Solidifiable a where
  compileObs :: Obs a -> State ObsCompileState T.Text
```

The `compileObs` function operates in the `State` monad, meaning that it can update the compile state. The return value contains the constructor for the top-level observable. Now consider what information an observable needs to expose: most obviously, its current value. However, making observables work for

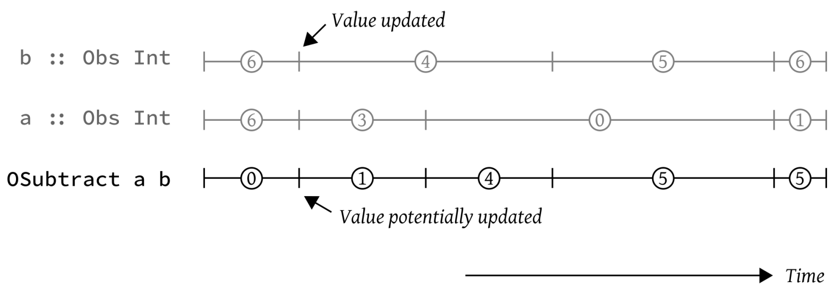

Figure 3.4: Combining the history of two observables.

primitives like *When* is more complicated. In these cases, we must be able to ascertain whether now is the *first* time (since a certain time) that the observable has fulfilled a certain condition.

To compute this, we store the history of the observable. Figure 3.4 shows how a composite observable might update based on its children.

Solidity does not yet have support for generic contracts or type parameters, so we will need a concrete version of the abstract base contract for each observable return type. At the moment, we only need boolean and integer observables. Here's what the abstract boolean observable looks like:

```
contract BoolObservable {
  struct Value {
    bool value;
    uint timestamp;
  }

  function getValueHistory() public returns(BoolObservable.Value[]);
  function getValue() public view returns(bool);
  function getTimestamp() public view returns(uint);

  function getFirstSince(function(bool) external pure returns(bool) condition, uint sinceTimestamp) public returns(
      bool, uint);
}
```

Notice that, for flexibility, `getFirstSince` takes a predicate rather than simply looking for the first time the observable was equal to a certain value.

With this in mind, we can start implementing the interpreter itself. For the `External` case, we can simply cast the provided address to the observable base class. There is no need to actually write any source code for the observable, as this is provided externally.

```
instance Solidifiable Bool where
  compileObs (External addr) = return [text|BoolObservable(${addr'})|]
    where addr' = T.pack addr
```

Next, a slightly more complicated example in the form of `Constant`. This is still fairly simple, as it has no children and only one historical value.

```
constantIntS :: Int -> Int -> (T.Text, T.Text)
constantIntS value idx = (
  [text|ConstantIntObservable_${idx'}|],
  [text|
    contract ConstantIntObservable_${idx'} is IntObservable {
      IntObservable.Value[] public valueHistory_;

      constructor() public {
        valueHistory_.push(IntObservable.Value(${value'}, 0));
      }

      function getValueHistory() public returns(IntObservable.Value[]) {
        return valueHistory_;
      }

      function getValue() public view returns(int) {
        return valueHistory_[0].value;
      }

      function getTimestamp() public view returns(uint) {
        return valueHistory_[0].timestamp;
```

```
    }

    function getFirstSince(function(int) external pure returns(bool) condition, uint) public returns(bool, uint) {
     if (condition(valueHistory_[0].value)) {
       return (true, 0);
     } else {
       return (false, 0);
     }
    }
  }
 |])
 where value' = showt value
 idx' = showt idx

instance Solidifiable Int where
 compileObs (Constant value) = do
  obsCounter += 1
  n ← use obsCounter
  let (name, source) = constantIntS value n
  obsSource %= addClass source
  return ([text|new ${name}()|])
```

The implementation of the various observable combinators is rather more difficult. I have omitted the full implementation here, but it is perhaps useful to see how `getValueHistory` and `getFirstSince` are implemented for `And`:

```
function getValueHistory() public returns(BoolObservable.Value[]) {
 BoolObservable.Value[] memory b1 = b1_.getValueHistory();
 BoolObservable.Value[] memory b2 = b2_.getValueHistory();
 valueHistory_.length = 0;

 uint i = 0;
 uint j = 0;

 while (i < b1.length && j < b2.length) {
  if (b1[i].timestamp < b2[j].timestamp) {
   valueHistory_.push(BoolObservable.Value(b1[i].value && b2[j-1].value, b1[i].timestamp));
   i++;
  } else if (b1[i].timestamp > b2[i].timestamp) {
   valueHistory_.push(BoolObservable.Value(b1[i-1].value && b2[j].value, b2[j].timestamp));
   j++;
  } else {
   valueHistory_.push(BoolObservable.Value(b1[i].value && b2[i].value, b1[i].timestamp));
   i++;
   j++;
  }
 }
 while (i < b1.length) {
  valueHistory_.push(BoolObservable.Value(b1[i].value && b2[j-1].value, b1[i].timestamp));
  i++;
 }
 while (j < b2.length) {
  valueHistory_.push(BoolObservable.Value(b1[i-1].value && b2[j].value, b2[j].timestamp));
  j++;
 }

 return valueHistory_;
}

function getFirstSince(function(bool) external pure returns(bool) condition, uint sinceTimestamp) public returns(
    bool, uint) {
 getValueHistory();
 uint currentTimestamp = 0;
 bool currentValue = valueHistory_[0].value;
 for (uint i = 0; i < valueHistory_.length; i++) {
  if (valueHistory_[i].timestamp < sinceTimestamp) {
   currentTimestamp = valueHistory_[i].timestamp;
   currentValue = valueHistory_[i].value;
   continue;
  } else {
   if (condition(currentValue)) {
    return (true, sinceTimestamp);
   }
   currentTimestamp = valueHistory_[i].timestamp;
```

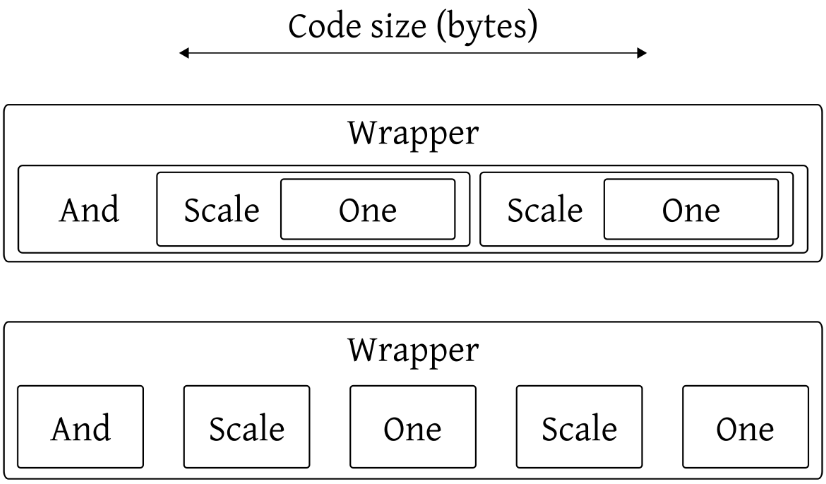

Figure 3.5: An illustration of code size pre- (top) and post-optimisation (bottom).

```
      currentValue = valueHistory_[i].value;
      if (condition(currentValue)) {
        return (true, currentTimestamp);
      }
    }
  }
  return (false, 0);
}
```

Clearly, this is not the most efficient of implementations. While the issue of observable history is complex to address, there is another efficiency issue that can be solved more easily.

3.9.5 Reducing code size

There are, of course, a wide range of optimisations that could be applied to our implementation. I will focus on one in particular: code size. Notice that our contracts instantiate their children like so:

```
Scale_0 next = new Scale_0(marketplace_, scale_, wrapper_, false, BoolObservable(0));
```

This means that each contract must *contain* all the the code for its subcontracts. As a result, code size is $O(n^2)$ in the number of primitives used. This is illustrated in Figure 3.5. We can address this problem by moving contract deployment into a method in the `WrapperContract`. At the cost of some overhead, this means that the wrapper size is $O(n)$ and each primitive is constant size. Now, contract instantiation looks like this:

```
contract WrapperContract {
  [...]
  function deploy(uint classId, [...]) public returns(BaseContract) {
    if (classId == 0) {
      return BaseContract(new Scale_0(marketplace, [...]));
    }

    if (classId == 1) [...]
  }
}

[...]

BaseContract next = wrapper_.deploy(0, marketplace_, scale_, wrapper_, false, BoolObservable(0));
```

A complex observable (e.g. `OAnd (OGreaterThan (External 0x123) (Constant 5)) (OAnd (After 1000) (Before 2000))`) suffers much the same problem. We can solve the problem again by moving observable deployment into the wrapper contract (I implemented the methods `deployIntObservable` and `deployBoolObservable`).

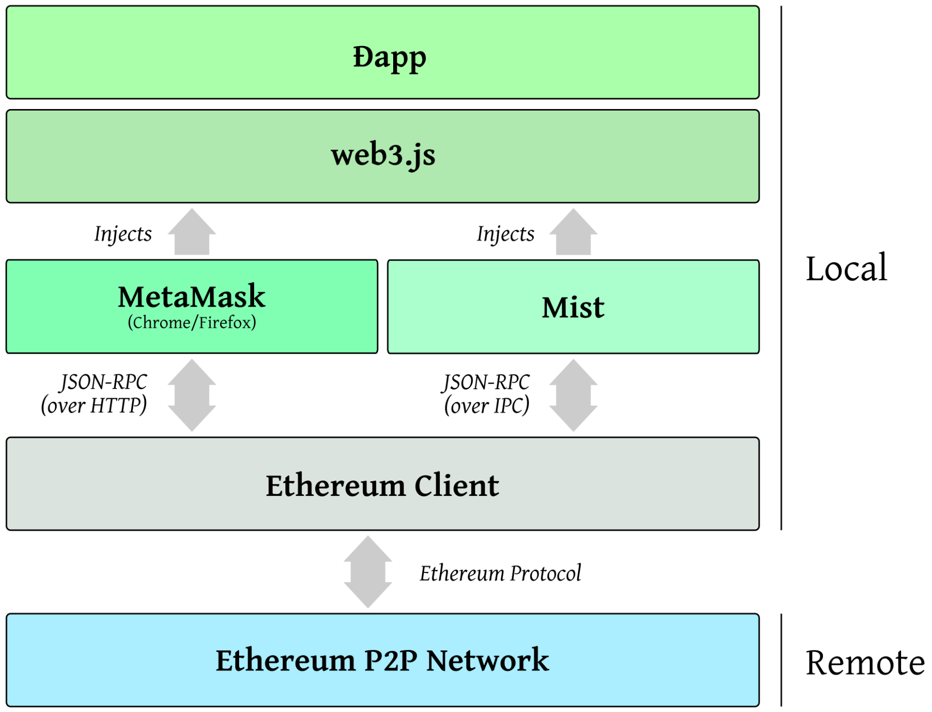

Figure 3.6: A common architecture for an Ethereum Đapp.

3.10 Deploying compiled contracts

To test the compiled contracts, I developed a web client, known as a *Đapp* in Ethereum terminology. This allows contracts to be uploaded and managed in a simple graphical environment. Due to the nature of Ethereum, the Đapp has a slightly complex architecture, shown in Figure 3.6. For development, I used the *go-ethereum* (geth) client. To avoid the cost of working on an actual blockchain, I also used geth simulate a local blockchain (testnet).

The Đapp (seen in Figure 3.7) is a Vue.js [4] app using the web3.js [5] library to query the Ethereum network. Users must access it through a browser (e.g. Mist) that injects a web3 instance. The browser itself it responsible for proxying requests made to web3 through to the local Ethereum client[7].

The Đapp has a number of features. Users can upload a *package* (containing a contract's bytecode and ABI definition), then deploy it to the blockchain. The Đapp allows users to propose, sign and proceed contracts, as well as managing the observables and marketplace on which they depend.

3.10.1 Deployment procedure

Merchant comes with a CLI that makes it easy to compile contracts into packages that can be deployed through the Đapp. I will briefly outline the process of running the Đapp, building a contract and executing it.

Prerequisites

Mist or an equivalent Đapp browser

geth tested with geth 1.8.6

npm tested with 5.8.0

Node.js tested with 9.5.0

solc tested with 0.4.23

Stack tested with 1.7.1

Set up the compiler

[7]This is normally done over a standard JSON-RPC interface.

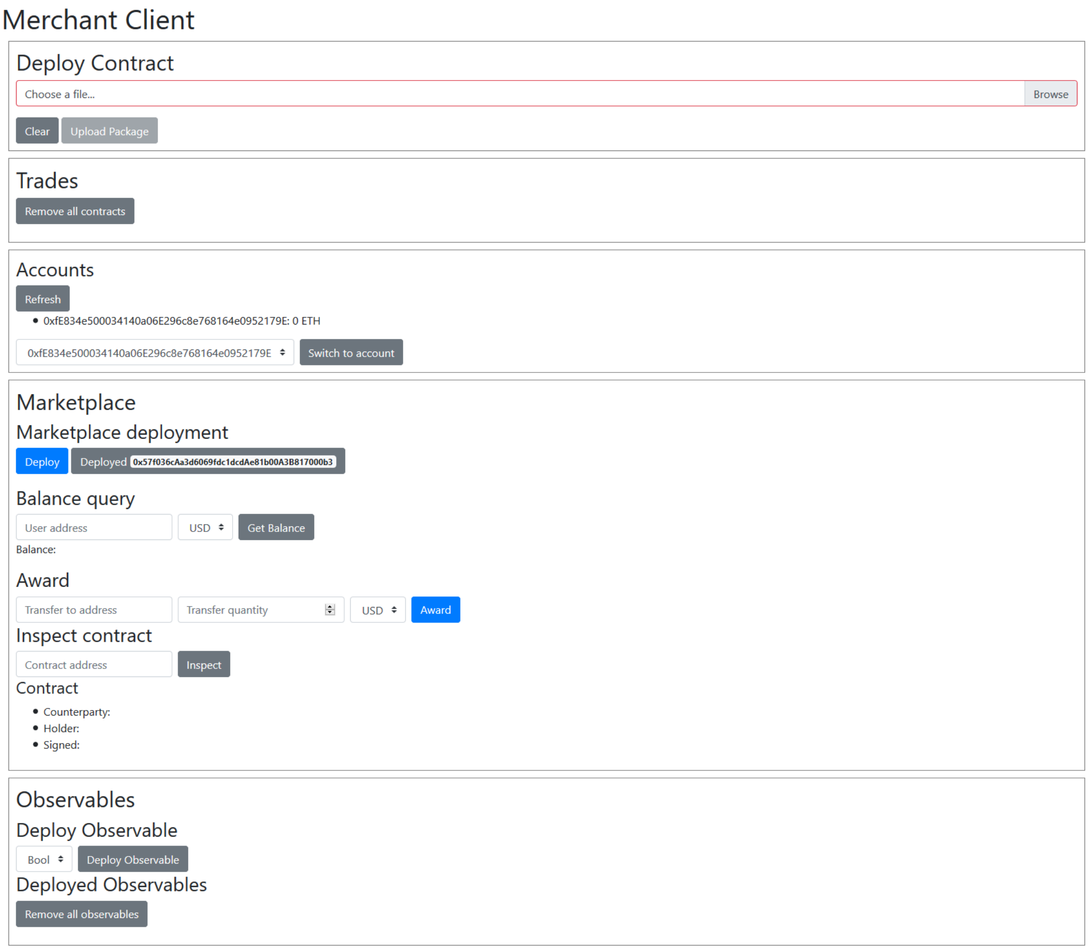

Figure 3.7: The Ðapp interface.

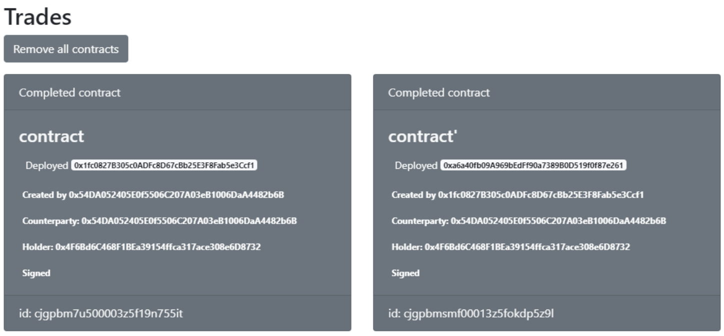

Figure 3.8: A view of some completed contracts in the Đapp.

Clone the repository `git clone https://github.com/rossng/merchant.git`

Build the compiler `cd merchant && stack build`

Set up the Đapp

Clone the repository `git clone https://github.com/rossng/merchant-client.git`

Install required packages `cd merchant-client && npm install`

Compile and run the Vue app `npm run dev`

Compile a contract

Write the contract Create a file, `contract.mc`, containing the contract `one' USD`.

Compile the contract `stack exec -- merchant compile --file contract.mc --package --output contract.json`

Deploy the contract

Launch geth For development, I suggest using `geth --dev`.

Launch Mist Make sure that you have created and funded a couple of Ethereum wallets. Running mist with the geth development mode gives you access to an account with a large amount of Ether by default.

Open the Đapp Go to `http://localhost:8080`.

Deploy a marketplace Click the *Deploy* button.

Upload the contract package Click *Browse* and find the `contract.json` package we compiled earlier.

Deploy the contract Click the *Deploy* button on the contract card.

Propose the contract Paste a wallet address into the *To address* field and click *Propose*.

Sign the contract Switch to the account to which you proposed the contract by choosing it from the dropdown list and clicking *Switch to account*. Click the now-enabled *Sign* button on the contract card.

Once a contract has been signed, execution will immediately proceed. As a contract executes, you should see the white cards (contracts that are currently alive) turn grey as they are executed and killed. The results of a completed contract can be seen in Figure 3.8. If there are decisions that need to be made at runtime (for the *Or* primitive), the interface will show the user buttons to change these.

Chapter 4

Evaluation

Now that we have explored the implementation of Merchant, I will evaluate the success of the project. First, I will look at the pros and cons of my implementation, and how my implementation choices have affected Merchant's usefulness. Second, I will compare Merchant to two alternatives: Findel and bespoke contracts. Finally, I will explore some potential areas for future work and assess whether Merchant is a useful platform to build on.

4.1 Functionality

These features are offered by the compiler and Ðapp in their current state:

- construction of contracts using primitives from both composing contracts papers;
- a modular, extensible DSL implementation;
- compilation of contracts to Solidity and deployable packages with a command line tool;
- deployment of compiled contracts, including the processes of proposing, signing and executing;
- and deployment and management of observable values.

However, there are some limitations:

- maximum contract size is currently very limited;
- the notion of the contract horizon is currently ignored by the extended set of contract primitives;
- there is no way for contracts to resume execution autonomously;
- and there is limited runtime information available in the UI about the execution state of the contract.

4.2 Implementation choices

4.2.1 Compilation output

One of my key implementation choices was how the declarative contracts are compiled to procedural Solidity code. Fundamentally, a contract can be seen as a nondeterministic finite state machine (NFA) like Figure 4.1.

I chose to simulate this NFA structure by using a Solidity `contract` to represent each state. To transition to a state, we simply instantiate it. Exiting a state means setting its `alive_` member to false. Epsilon transitions happen when contracts immediately call `proceed` on their newly instantiated children.

This is by no means the only way of simulating an NFA. Another approach, for example, would be to use a single `contract` that maintains a list of currently active states in the state machine. When `proceed` is called, it would use a series of `if`-statements to choose which part of the contract's logic to execute.

I implemented the first approach for simplicity's sake, but it is likely that the second approach (or something similar) would be more efficient. The cost of creating a contract is at least 32000 gas—the single most expensive operation in Ethereum—so we probably want to avoid it.

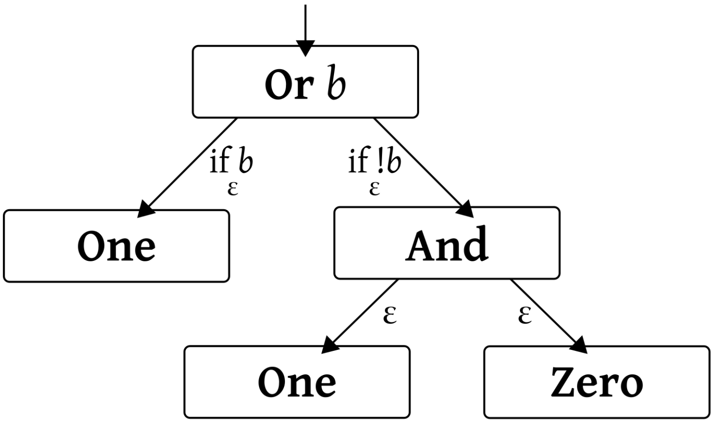

Figure 4.1: An NFA representing the contract *Or(One k, And(One k, Zero))*.

4.2.2 Handling 'autonomous' primitives

One of the trickiest problems I faced in the implementation was dealing with contract primitives such as *When* and *Get*. In fact, it is simply impossible to express the true intent of these primitives in Ethereum, because code execution must be initiated externally.

There are a number of partial solutions to this problem. I chose to treat a *failure to proceed the contract* as a mutual agreement to void the contract. A more ideal solution would be to somehow guarantee that the contract will execute at the correct time. There are a few ways to approach this:

- Require one party to proceed the contract at the correct time, else face a penalty in the contract. While this does not guarantee execution, it does have deterministic consequences if the contract fails to proceed.
- Use an escrow to proceed the contract. It might be possible to use a trusted execution environment to prove that the escrow is running the correct software. This, however, is still vulnerable to network partitions, power loss or bad actors.
- Use a service like the Ethereum Alarm Clock, which uses a reward to incentivise others to call the contract at the correct time. Again, this does not *guarantee* execution.

4.2.3 Commodity balances and debt

Another design choice was to use a marketplace contract that simply stores accounts' commodity balances as an integer. This was done for simplicity, but most Ethereum-based 'tokens' instead conform to the ERC-20 standard [45].

This also means that balances can be less than zero—there is no concept of debt, or enforcement of debt. As I highlighted in subsection 2.3.3, the enforcement of debt in Ethereum is a non-trivial and unsolved problem. Because debt is not the core focus of this project, I have deliberately left it open as a future research topic.

4.2.4 Observable interpreter

I had originally planned to implement the observables and their interpreter in a free monadic style, much like the other contract primitives. However, the free type parameter of `Obs a` makes this currently impossible.

Instead, I opted to use a more standard, generally-recursive approach to embedding and interpreting observables. Though this makes the implementation less consistent, it does work.

4.2.5 Observable history

The current implementation of observable values requires every observable to store its entire history, which can be very expensive. This is a workaround for Ethereum's lack of contract-triggering events. That is, a contract primitive such as *When* cannot 'subscribe' to observable updates and start execution when its condition is met. In order for a primitive to determine whether to execute, an observable must be able to report whether now is the *first* time that a condition has been met.

There are a few potential workarounds for this problem, which I will talk about in Section 4.4.

4.3 Comparison to alternatives

4.3.1 Findel

Findel is the most comparable system, embedding a composing contracts-esque language directly into the Solidity language. Findel has various advantages and disadvantages compared to Merchant.

Ease of deployment Only the marketplace contract needs to be deployed. Creation of a Findel contract simply requires calling a number of methods on the marketplace, whereas Merchant requires contracts to be deployed.

Simplicity There is no need for a special-purpose compiler, as the Findel language is a deep embedding in Solidity.

Observable support It is not possible to express observables in the contract definitions. Findel has separate `Scale` and `ScaleObs` primitives because it cannot express the concept of a constant observable value. Findel only supports 'gateways'—the equivalent of an `External` observable—but they are not type safe.

Extensibility The available primitives are defined in the marketplace contract. Introducing new primitives requires the marketplace to be updated; our approach does not necessitate this in most cases. Our extensible DSL embedding makes it easy to add new primitives to Merchant. In addition, this makes it possible to optimise contracts by combining primitives—something which Findel cannot support.

Expressiveness Findel does not implement primitives such as *Get* or *When*, and its replacements are not semantically equivalent. Our implementation reflects the original composing contracts language much more closely.

Tooling The tooling provided with Findel is tricky to set up and use, and only works with a local blockchain by default. The Merchant Ðapp and tools are designed to be easy to use and chain-agnostic.

Execution cost

Another interesting comparison between the Merchant and Findel approaches is the relative cost of executing contracts and other operations, shown in Table 4.1. The different design approaches of the two projects have different implications, including varying cost of execution. I compare the execution models of both systems in more detail in Appendix A.

While recording these measurements, I made several contributions to the original Findel codebase and associated tools. In particular, I updated Findel's marketplace contract[1] to meet the latest Solidity language standard (0.4.23 as of May 2018).

The original Findel paper lists a number of sample contracts which were used for benchmarks. However, I discovered that it was not possible to deploy many of these using Merchant because of code size limitations currently imposed by the Ethereum protocol. EIP-170 [46] limits contract code deployments to a maximum of 24576 bytes. In addition, blocks on the main Ethereum chain are limited to using approximately eight million gas. While we can avoid the gas limit by using a local blockchain for testing, the EIP-170 limit is much harder to disable.

As of May 2018, the price of one Ether is about 530 GBP. Additionally, the cost per unit gas is approximately 2 Gwei (2×10^{-9} Ether). The cost to deploy the Findel marketplace, for example, is 2×10^{-9} Ether gas^{-1} $\times$ 530 GBP Ether^{-1} $\times$ 1797270 gas = 1.91GBP. I used Remix, an in-browser Solidity compiler and blockchain simulator, to compile, deploy and measure the contracts. All contracts were compiled with the Solidity compiler `--optimize` flag enabled. Remix reports both transaction cost and execution cost.

[1] https://github.com/cryptolu/findel/pull/12

Operation	Transaction cost	
	Findel	Merchant
Deploy the marketplace	1797270	**956859**
Registering with marketplace	102378	**0** (N/A)
Create and propose/issue *One(USD)*	**251543**	1257282
Execute *One(USD)*	**53017**	611631
Create and propose/issue *zero-coupon bond*	**494947**	3914897
Sign *zero-coupon bond* before t	N/A	1323804
Later resume execution of *zero-coupon bond*	53017	28242

Table 4.1: A comparison of the cost of different operations using Findel and Merchant.

Conclusion

There is a key tension in the design of both systems.

Findel allows us to deploy larger contracts, but their behaviour is limited to that already available in the marketplace. The marketplace has a bytecode size of approximately 6300 bytes, so there is still some (but finite) room to add additional functionality. Observable support is very limited.

Merchant (in its current form) does not allow deployment of large contracts, because there is a significant overhead for each primitive used. However, contract behaviour is modular and extensible—and crucially, new primitives can be introduced without adding new logic to the marketplace contract (currently about 3400 bytes).

I will address some potential solutions to this problem in Section 4.4. For now, Findel remains a more 'practical' system than Merchant. However, neither system is truly usable without some solution for debt enforcement, so this is a moot point. Merchant provides a more advanced language and is designed to support multiple backends, so I believe it is a better platform for further exploration.

4.3.2 Hand-crafted contracts

Another approach to building financial contracts for Ethereum is to write bespoke contracts for each trade or each type of trade. This has a number of implications:

Flexibility The traditional approach to contract creation in Ethereum is to manually author contracts in Solidity itself. The key advantage of this approach is that it is extremely flexible: contracts can include arbitrary Turing-complete computation, allowing them to model very complex scenarios. However, they are still bound by some of the same constraints as our compiler—for example, contracts cannot cause themselves to be called when an event happens.

Efficiency Another possible advantage is increased efficiency—contract code can reflect the exact intention of the author, minimising the gas cost of contract execution. However, there is a strong argument to be made that an optimising Merchant compiler could eventually exceed the average efficiency of bespoke contracts with much less human effort.

Correctness Manual authoring of contracts has various disadvantages. Chief among them is correctness: it is easy to write subtly incorrect contracts that allow attackers to tamper with execution. A good example is the Decentralised Autonomous Organisation (DAO) [47]. This contract was meant to act as a democratic, decentralised investment organisation, and was funded with about twelve million Ether. A flaw in the contract design was quickly found, and an attacker managed to steal over three million Ether [48]. This attack was so successful that the Ethereum blockchain was eventually forked[2].

Similar costly errors have occurred multiple times due to faulty smart contracts. This is arguably inevitable when contracts are written in an unconstrained language like Solidity: it is exceptionally

[2] The official chain—where the attack was reversed—is now known as ETH, and the unmodified chain as ETC.

difficult and time-consuming to prove correctness. In the declarative composing contracts language, however, developers are prevented from using any unsafe behaviour. In addition, the intent of contracts should be clear from how they are written—an argument which cannot be made for non-trivial Solidity programs.

Of course, we should prove our compiler in order to have full confidence in the correctness of compiled contracts. Doing so exceeds the scope of this project, but should be a considerably easier task than proving the correctness of each bespoke contract.

Time-to-market Another benefit of a declarative contract language is that it considerably reduces the time cost and difficulty of authoring usable financial contracts. The language allows creation and composition of high-level combinators representing widely-understood financial instruments (e.g. options, swaps, futures). This means that contracts can potentially be authored by financial—rather than technical—experts.

Frankau et al. at Barclays Capital wrote about their experience of migrating exotic financial derivatives to a composing contracts-inspired platform [49]. Here is how they describe the legacy process:

> Quants define a standardized pattern to represent a trade type [...]. Risk managers sign off the library, and then the IT department develops a front-end [...]. Each trade is saved in an ad hoc serialization format. [...] Since new templates are so expensive there can be a tendency to cram a new trade type into an existing structure through creative misuse of existing features, increasing the risk of mistakes.

The new system was designed with a 'small set of core primitives', the intention being to avoid the arduous template-creation process while still permitting users to create new kinds of exotic trades. According to the authors, this has been a success:

> We now write all new trades using FPF. In addition, we have migrated all our one-off trades to FPF, freeing up reserve cash devoted for operational risk. Currently we are also starting to migrate templated trades to FPF. [...] [W]e reduced the development cycle.

A similar comparison can be made between bespoke smart contracts and declarative contract DSLs. Merchant allows the creation of new financial contract designs with a stable core infrastructure. New types of bespoke contract, however, often require infrastructural changes.

Conclusion

There is a clear benefit to a Merchant-like system compared to writing contracts by hand. When dealing with large amounts of money, correctness is very important. It is likely, too, that Merchant-style contracts could be just as efficient as bespoke contracts. These two attributes make it an appealing approach for any party considering blockchain-mediated financial trades.

4.4 Further work

There are a number of potential areas for further work.

4.4.1 Correctness

Before using Merchant in a live environment, it would be desirable to prove the correctness of the compiler and the marketplace contract. It would also be useful to add a test suite that covers a set of example contracts, but the tools for doing this are currently very limited. In future it may be possible to use a fuzz testing tool like Echidna [50]. Popular Đapp frameworks (e.g. Truffle, Embark) do include test frameworks, but their architecture is fundamentally incompatible with Merchant.

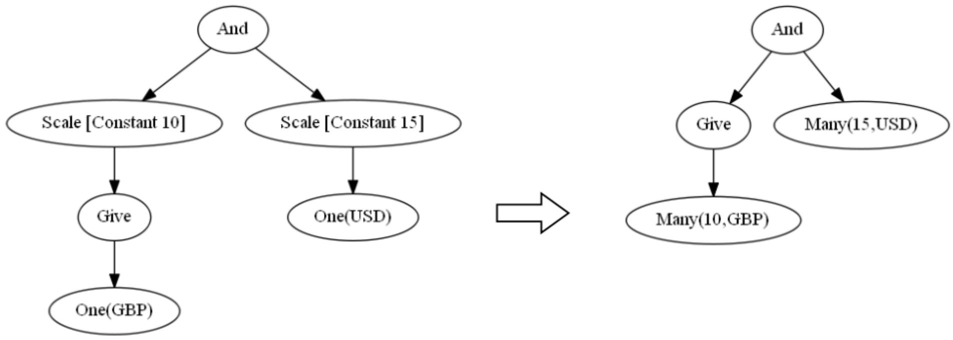

Figure 4.2: An example of optimising a contract AST by rewriting it use the *Many* primitive.

4.4.2 Observables

Better implementation

Future work should make it possible to implement observables in a free monadic style. Reimplementing the observable language like this will make it extensible, which would be very useful. For example, it would reduce the severity of the Expression Problem when adding new observable operators.

In addition, the entire language could be migrated to the *compdata* library described by Bahr and Hvitved [51]. This builds on the ideas on Swierstra's Data Types á la Carte [11], but adds performance improvements, more usable modularity and a suite of tools for working with compositional data types.

Provable observables

The Findel paper suggests that observable 'gateways' can be queried for a proof of their current value. A common use of external observables is to make 'off-chain' data available inside the blockchain. However, this currently relies on a trusted party. Future work could leverage public key infrastructure (PKI) to solve this—for example, market operators could provide cryptographically signed market data to be made available on the blockchain.

Efficiency

The current implementation of observables in Merchant requires storage of their entire value history. For observables that change frequently (e.g. temperature, current time) this is untenable. For this reason, Merchant currently uses *At*, *Before* and *After* for time-based observables, rather than the more flexible *Date* that is suggested in the original papers.

One solution may be to give observables knowledge of the contract primitive that depends on them. For example, *External* observables (those updated by a user or third-party) could store a list of contract callbacks and conditions: when a condition becomes true for the first time, the callback is executed. Alternatively, with knowledge of the contract primitives that use them, it may be possible to avoid keeping observable history in most instances.

These improvements may be tricky to implement, but would likely provide a large efficiency improvement.

4.4.3 Optimisation

There is considerable scope to optimise the contracts produced by Merchant.

Rewriting the AST

First, it should be possible to optimise the AST into considerably fewer operations. For example, a common pattern is to use a *Scale(Constant(n), One(currency))* to cause the contract holder to receive n units of *currency*. This could be easily simplified to a single primitive *Many(n, currency)*. An example of this optimisation is shown in Figure 4.2. Notice that we can even move the *Scale* primitive across the *Give*—the two primitives commute without changing the contract's meaning.

Here, we benefit from having a modular DSL. Without changing the interface exposed to the user, we can add our new *Many* primitive:

```haskell
data OptimiseF next
  = Many Int Currency
  deriving (Functor)
```

We can also add a type synonym for *internal* contracts—i.e. ones that our compiler should handle, but which include primitives not available to contract authors.

```haskell
type ContractInternal = Free (ContractF :+ OriginalF :+ ExtendedF :+ OptimiseF) ()
```

At this point, we need to do two things to make use of our optimisation. The first is to add a Solidity compiler algebra for the *Many* primitive—simply a case of implementing `instance SolidityAlg OptimiseF`. The second is to implement an optimisation step where we rewrite the AST of our contract, converting *Scale(One)* constructs into *Many*s. Having a `Free` AST means that we can do this with recursion schemes (as described by Meijer [52]), possibly through Kmett's *recursion-schemes* library.

Here are a few of the other potential optimisations:

- Combining the children of *And* (and particularly chains of *And*s).
- Compile-time evaluation of statically-known observables (or subtrees of observables) to a single constant.
- Combining separate subtrees that are gated by the same condition (e.g. *When*).

Rethinking the compiler

When implementing code generation in Merchant I chose simplicity over efficiency. However, there are many alternative ways to implement the compiler.

One potential approach is more Findel-like. By embedding the behaviour of contract primitives directly into the marketplace, it may be possible to avoid the overhead of repeatedly deploying this behaviour for each new contract. This could be done in an extensible way, with some behaviour integrated into the marketplace and some deployed on a per-contract basis.

Another approach is to deploy a single Solidity contract for each Merchant contract. The contract behaviour would then be implemented as a state machine—calls to `proceed` would simply update some internal state rather than spawning new sub-contracts. This would reduce overall code size, but the contracts would soon approach the gas and code size limits.

Another option worth considering is to split contract deployment into multiple transactions. Though this would not decrease the overall cost of deploying and executing a contract, it would make it possible to deploy very large contracts without falling foul of Ethereum's limits.

There is an existing field of research into the minimisation of finite automata [53]. Merchant contracts can be considered a minor variation on nondeterministic finite automata (NFAs), and it may be possible to apply existing techniques to merge states. This would reduce code size and execution cost.

Finally, further work could explore a different compilation target. Compiling to Solidity currently requires us to translate our contract semantics into Solidity's object-oriented semantics. The Solidity compiler then attempts to automatically translate these yet again into EVM bytecode. It may be possible to directly generate efficient EVM bytecode using our semantic knowledge of how Merchant contracts are structured. This, however, would require significant development effort.

4.4.4 Debt

As discussed, debt enforcement remains a complex and unsolved issue. Future work that addresses this problem will be widely applicable to many blockchain systems.

4.4.5 Valuation

The original composing contracts papers [1], [17] describe a semantics for valuing contracts. They define a denotational semantics which can be used to compute the expected value of any contract, then briefly describe a concrete implementation that uses a lattice model [54].

It would certainly be useful to be able to compute the expected value of one's Ethereum contracts, so I investigated reimplementing the lattice model as part of this project. However, I decided to leave it out-of-scope: financial modelling is complex and many companies have spent significant money on building their own proprietary solutions.

4.4.6 More powerful contracts

There is scope to expand the power of Merchant contracts in several axes. Most obviously, there may be useful new primitives that can be added to the language. It should be easy to do this work on top of the existing Merchant implementation, which is explicitly designed to be extensible.

Second, further work could look at adding new contract paradigms. For example, cyclic contracts would be useful for situations where contract duration is unbounded or where the contract repeatedly performs an action (e.g. swing contracts for gas supply). Other useful extensions might include contracts with more than two parties.

It would also be useful for the compiler implementation to permit contracts to perform IO actions at compile time. For example, a contract author should be able to retrieve the current time and use offsets from it inside a contract definition.

4.5 Summary

We can compare Merchant to its original technical aims:

Reimplement composing contracts using modern Haskell This has been mostly successful, with the (partial) exception of observables. The use of free monadic data structures gives us a number of opportunities to improve Merchant further (e.g. through AST manipulation with recursion schemes).

Adapt the language as necessary I have successfully translated all of the original composing contracts primitives to Merchant. While there have been some compromises (e.g. mutual voiding of *When*), the primitives generally retain their original semantic intent.

Build a prototype client I have successfully built a Ðapp that allows users to deploy and manage their contracts.

Evaluate usability In this chapter, I have compared Merchant against Findel and bespoke contracts, finding a number of pros and cons.

Reduce execution cost I have implemented some simple optimisations to reduce code size (and hence the cost of deploying and executing contracts). In addition, I have highlighted a range of possible future improvements to Merchant.

While Merchant remains experimental, I have been able to achieve (or nearly achieve) the technical goals of the project. Compared to other options, Merchant is an attractive platform for future research.

Chapter 5

Conclusion

The core aims of this project were to *reimplement composing contracts using modern Haskell* and to *write a compiler that converts the composing contracts language to Ethereum smart contracts*. While there are some minor incompatibilities between the semantics of Ethereum and composing contracts, I have been able to achieve this to a large degree. Using the compiler and Ðapp developed for this project, it is possible to write, compile and execute a variety of financial contracts. Some examples of contracts can be seen in Appendix B.

It is fair to say that a number of major issues remain before this technology can be deployed in real-world scenarios. Chiefly, these are: verification of the compiler; enforcement of debt and high execution costs. I have not encountered any evidence suggesting that these are insurmountable issues. Solving them simply requires more development time.

At a more abstract level, I postulated that smart contracts could enable *reliable*, *fast* and *transparent* financial markets. Having shown that we can represent traditional financial instruments on a decentralised blockchain, I hope that this is one step closer to reality.

Reliability of the market is a property that is mostly delegated to the Ethereum protocol itself. However, I have encountered some interesting reliability issues, particularly the question of ensuring that contracts resume execution when some condition is fulfilled. While I have implemented a simple solution, this remains an open question.

Transparency is achieved by this project in the sense that all transactions are visible and deterministic. However, Ethereum addresses are not tied to public identities. This is connected to the issue of debt enforcement, for which solutions will likely require connecting real-world identities to Ethereum accounts. It remains to be seen whether solutions to the debt problem will result in full transaction transparency, or whether pseudonymity will be retained.

Speed and efficiency remain the major issues. Financial markets operate on the scale of milliseconds, but blockchain protocols cannot currently compete with these speeds. In addition, operating at the scale of financial markets will require many improvements to blockchain technology. Today, the maximum throughput of the Ethereum blockchain is approximately 15 transactions per second. By contrast, the NYSE Group processes around 200 trades per second, and Visa around 2000. This is an ongoing area of research [55]. The cost of executing Merchant contracts is currently high, but there is no reason to believe that it could not be significantly reduced through compiler improvements.

To sum up: this project has successfully developed Merchant, a system for declarative authoring of financial contracts and deployment of those contracts to the Ethereum blockchain. This has clarified what financial constructs can easily be expressed on a blockchain, and which cannot. Overall, however, I have shown that most contract semantics can be readily translated for execution on a blockchain. I have also discussed the wide range of research and development work that remains before a Merchant-like system can be used in production.

Bibliography

[1] S. Peyton Jones, J.-M. Eber and J. Seward, “Composing contracts: An adventure in financial engineering (functional pearl)”, in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP ’00, New York, NY, USA: ACM, 2000, pp. 280–292, ISBN: 1-58113-202-6. DOI: 10.1145/351240.351267. [Online]. Available: http://doi.acm.org/10.1145/351240.351267.

[2] E. Kmett, *Free*, 2018. [Online]. Available: https://hackage.haskell.org/package/free.

[3] M. Sackman and I. L. Miljenovic, *Graphviz*, 2017. [Online]. Available: http://hackage.haskell.org/package/graphviz.

[4] E. You, *Vue.js*, 2014/2018. [Online]. Available: https://github.com/vuejs/vue.

[5] *Web3.js*, 2014/2018. [Online]. Available: https://github.com/ethereum/web3.js.

[6] D. Cliff and J. Bruten, “Simple bargaining agents for decentralized market-based control”, 1998.

[7] D. du Preez, *How LSE was stricken by Millennium bugs*, 2011. [Online]. Available: https://www.computing.co.uk/ctg/analysis/2032112/lse-stricken-millennium-bugs.

[8] Euronext, *Introduction of T+2 standard settlement lifecycle*, Feb. 2014. [Online]. Available: https://www.euronext.com/sites/www.euronext.com/files/140111introt2settlementcycleinfoflash.pdf.

[9] P. Stafford, *UK share settlement to be cut to two days*, Dec. 2013. [Online]. Available: https://www.ft.com/content/67f95f76-5b56-11e3-848e-00144feabdc0.

[10] “Structuring, pricing and processing complex financial products with MLFi”, Tech. Rep. [Online]. Available: http://www.lexifi.com/files/resources/MLFiWhitePaper.pdf (visited on 18/03/2018).

[11] W. Swierstra, “Data types à la carte”, *J. Funct. Program.*, vol. 18, no. 4, pp. 423–436, Jul. 2008, ISSN: 0956-7968. DOI: 10.1017/S0956796808006758. [Online]. Available: http://dx.doi.org/10.1017/S0956796808006758.

[12] Aristotle, *Politics*, Greek and English, trans. by H. Rackham. Cambridge, MA and London: Harvard University Press and William Heinemann Ltd., 1944. [Online]. Available: http://www.perseus.tufts.edu/hopper/text?doc=urn:cts:greekLit:tlg0086.tlg035.perseus-eng1:1.1259a.

[13] “IAS 32—financial instruments: Presentation”, Tech. Rep., 2003, ch. 10. [Online]. Available: http://eifrs.ifrs.org/eifrs/bnstandards/en/IAS32.pdf.

[14] D. A. Moss and E. Kintgen, *The Dojima Rice Market and the Origins of Futures Trading*. Harvard Business School, 2010. [Online]. Available: https://hbr.org/product/the-dojima-rice-market-and-the-origins-of-futures-trading/709044-PDF-ENG.

[15] S. Agarwal, K. Xu and J. Moghtader, “Toward machine-understandable contracts”, *AI4J–Artificial Intelligence for Justice*, p. 1, 2016.

[16] B. Arnold, A. van Deursen and M. Res, “An algebraic specification of a language for describing financial products”, in *ICSE-17 workshop on formal methods application in software engineering*, 1995, pp. 6–13.

[17] J. Gibbons and O. d. Moor, *The fun of programming*. Palgrave Macmillan, 2003, ISBN: 9780333992852.

[18] N. Szabo, “Smart contracts: Building blocks for digital markets”, 1996. [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[19] S. Haber and W. S. Stornetta, “How to time-stamp a digital document”, *J. Cryptol.*, vol. 3, no. 2, pp. 99–111, Jan. 1991, ISSN: 0933-2790. DOI: 10.1007/BF00196791. [Online]. Available: http://dx.doi.org/10.1007/BF00196791.

[20] D. Bayer, S. Haber and W. S. Stornetta, "Improving the efficiency and reliability of digital time-stamping", in *Sequences II*, R. Capocelli, A. De Santis and U. Vaccaro, Eds., New York, NY: Springer New York, 1993, pp. 329–334, ISBN: 978-1-4613-9323-8.

[21] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail", in *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '92, London, UK, UK: Springer-Verlag, 1993, pp. 139–147, ISBN: 3-540-57340-2. [Online]. Available: http://dl.acm.org/citation.cfm?id=646757.705669.

[22] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols", in *Proceedings of the IFIP TC6/TC11 Joint Working Conference on Secure Information Networks: Communications and Multimedia Security*, ser. CMS '99, Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 1999, pp. 258–272, ISBN: 0-7923-8600-0. [Online]. Available: http://dl.acm.org/citation.cfm?id=647800.757199.

[23] W. Dai, "B-money", 1998. [Online]. Available: http://www.weidai.com/bmoney.txt.

[24] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", 2008.

[25] J. Matonis, *The bitcoin mining arms race: Ghash.io and the 51% issue*, 2014. [Online]. Available: https://www.coindesk.com/bitcoin-mining-detente-ghash-io-51-issue/.

[26] "Ethereum white paper", [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper/43bb8ec3fedec0192a5a796577f8b6104f3d675f#ethereum-accounts.

[27] *Introduction to smart contracts.* [Online]. Available: http://solidity.readthedocs.io/en/v0.4.22/introduction-to-smart-contracts.html#storage.

[28] *Solidity by example.* [Online]. Available: http://solidity.readthedocs.io/en/v0.4.22/solidity-by-example.html#simple-open-auction.

[29] A. Josang and R. Ismail, "The beta reputation system", in *Proceedings of the 15th bled electronic commerce conference*, vol. 5, 2002, pp. 2502–2511.

[30] R. Dennis and G. Owen, "Rep on the block: A next generation reputation system based on the blockchain", in *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, Dec. 2015, pp. 131–138. DOI: 10.1109/ICITST.2015.7412073.

[31] M. Srivatsa, L. Xiong and L. Liu, "TrustGuard: Countering vulnerabilities in reputation management for decentralized overlay networks", in *Proceedings of the 14th International Conference on World Wide Web*, ser. WWW '05, Chiba, Japan: ACM, 2005, pp. 422–431, ISBN: 1-59593-046-9. DOI: 10.1145/1060745.1060808. [Online]. Available: http://doi.acm.org/10.1145/1060745.1060808.

[32] C. Georgen, *Topl, Whitepaper, version 1.1*, 2017. [Online]. Available: https://topl.co/whitepaper.

[33] *The USN*, 2018. [Online]. Available: https://slock.it/usn.html.

[34] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger, Byzantium version d762c36", Apr. 2018. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.

[35] M. Gordon, "Linking ACL2 and HOL: Past achievements and future prospects", in *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014.*, 2014.

[36] R. J. Boulton, A. D. Gordon, M. J. Gordon, J. Harrison, J. Herbert and J. Van Tassel, "Experience with embedding hardware description languages in HOL", in *TPCD*, vol. 10, 1992, pp. 129–156.

[37] Netrium.org Contributors, *Netrium*, 2018. [Online]. Available: https://github.com/netrium/Netrium.

[38] A. Biryukov, D. Khovratovich and S. Tikhomirov, "Findel: Secure derivative contracts for ethereum", in *Financial Cryptography and Data Security*, K. Brenner Michael and Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala and M. Pintore Federico and Jakobsson, Eds., Cham: Springer International Publishing, 2017, pp. 453–467, ISBN: 978-3-319-70278-0.

[39] P. Wadler, "The expression problem", *Java-genericity mailing list*, 1998.

[40] M. Torgersen, "The expression problem revisited", in *ECOOP 2004 – Object-Oriented Programming*, M. Odersky, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 123–146, ISBN: 978-3-540-24851-4.

[41] P. M. Cohn and P. M. Cohn, *Universal algebra*. Reidel Dordrecht, 1981, vol. 159.

[42] S. Mac Lane, *Categories for the working mathematician*. Springer, 1998.

[43] P. JF, Nov. 2012. [Online]. Available: https://stackoverflow.com/a/13357359/.

[44] C. McBride, "Functional pearl: Kleisli arrows of outrageous fortune",

[45] F. Vogelsteller and V. Buterin, *Eip 20: Erc-20 token standard*, Nov. 2015. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[46] V. Buterin, *Eip 170: Contract code size limit*, Nov. 2016. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-170.md.

[47] ——, *CRITICAL UPDATE Re: DAO Vulnerability*, Jun. 2016. [Online]. Available: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/.

[48] *Bitcoin and Beyond Cryptocurrencies, Blockchains, and Global Governance*. Routledge, 2018. [Online]. Available: https://www.taylorfrancis.com/books/e/9781351814089.

[49] S. Frankau, D. Spinellis, N. Nassuphis and C. Burgard, "Commercial uses: Going functional on exotic trades", *Journal of Functional Programming*, vol. 19, no. 1, pp. 27–45, 2009. DOI: 10.1017/S0956796808007016.

[50] J. Smith, *Echidna*, 2017/2018. [Online]. Available: https://github.com/trailofbits/echidna.

[51] P. Bahr and T. Hvitved, "Compositional data types", in *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, ser. WGP '11, Tokyo, Japan: ACM, 2011, pp. 83–94, ISBN: 978-1-4503-0861-8. DOI: 10.1145/2036918.2036930. [Online]. Available: http://doi.acm.org/10.1145/2036918.2036930.

[52] E. Meijer, M. Fokkinga and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire", Springer-Verlag, 1991, pp. 124–144.

[53] T. Kameda and P. Weiner, "On the state minimization of nondeterministic finite automata", *IEEE Transactions on Computers*, vol. C-19, no. 7, pp. 617–627, Jul. 1970, ISSN: 0018-9340. DOI: 10.1109/T-C.1970.222994.

[54] T. S. Y. Ho and S. Lee, "Term structure movements and pricing interest rate contingent claims", *The Journal of Finance*, vol. 41, no. 5, pp. 1011–1029, DOI: 10.1111/j.1540-6261.1986.tb02528.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1540-6261.1986.tb02528.x. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1540-6261.1986.tb02528.x.

[55] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song and R. Wattenhofer, "On scaling decentralized blockchains", in *Financial Cryptography and Data Security*, J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner and K. Rohloff, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 106–125, ISBN: 978-3-662-53357-4.

Appendix A

Contract execution models

Figures A.1 and A.2 show the execution models of Merchant and Findel respectively. Some interesting notes:

- Merchant pushes most of the contract creation process off the blockchain and into the compiler. Findel does contract construction directly inside the marketplace contract itself.
- Merchant delegates contract logic into independent contract objects. Findel stores and executes all contract logic inside the marketplace.
- Both systems have a similar three step process:

Operation	**Findel**	**Merchant**
Creation	User calls marketplace functions and finally `createFincontract` to build up contract description.	User compiles DSL and deploys resulting smart contract to blockchain.
Proposal	User calls `issueFor`.	User calls `propose`.
Acceptance	Other party calls `join` and contract executes.	Other party calls `sign` and contract executes.

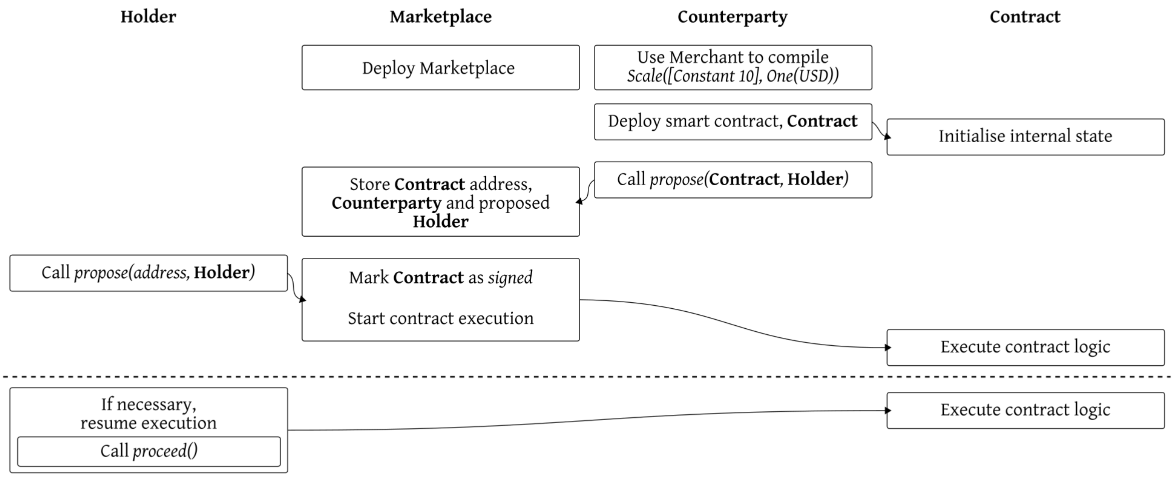

Figure A.1: The flow of creating and executing a Merchant contract.

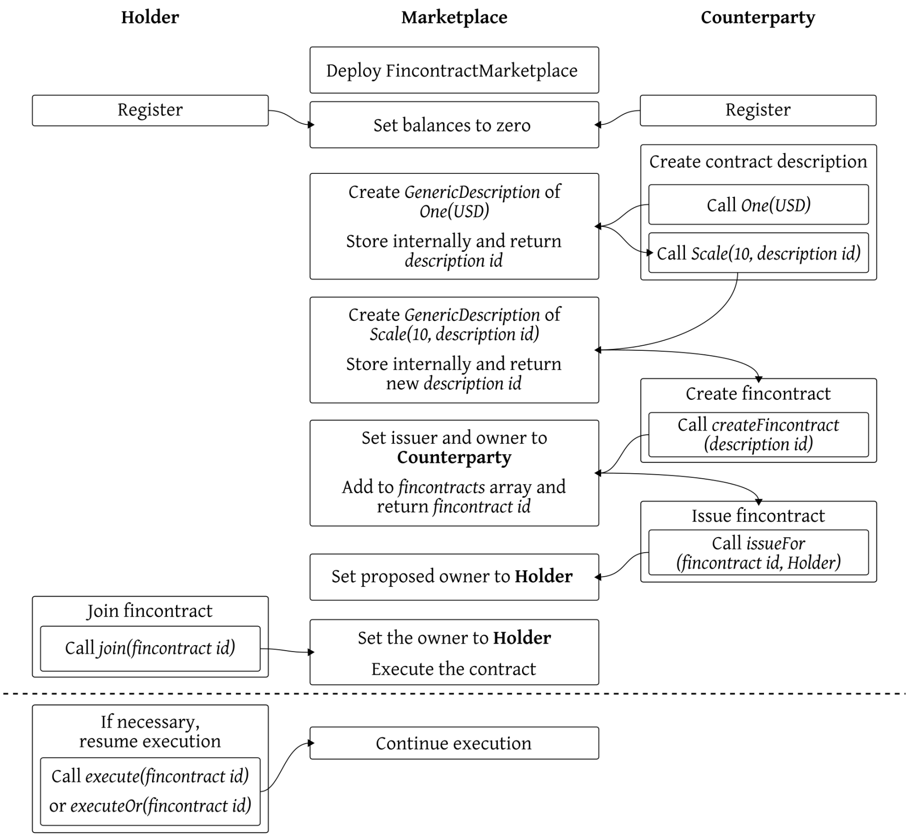

Figure A.2: The flow of creating and executing a Findel contract.

Appendix B

Example contracts

Examples adapted from Findel [38].

Fixed-rate currency exchange
10 USD is purchased by the holder for 7 GBP.

Merchant `and' (give' (scaleK' 7 (one' GBP))) (scaleK' 10 (one' USD))`

Merchant (do-notation) `and' (do {giveM; scaleKM 7; oneM GBP}) (do {scaleKM 10; oneM USD})`

Findel `And(Give(Scale(7, One(GBP))), Scale(10, One(USD))`

Zero-coupon bond
The holder receives 100 USD at time `t`.

Merchant `when' (At t) (scaleK' 100 (one' USD))`

Findel `Timebound(t -` δ`, t +` δ`, Scale(100, One(USD)))`

Bond with coupons
The holder receives a 10% coupon of 50 GBP for two years (`t1` and `t2`), then the face value of 500 GBP after three years (`t3`).

Merchant `and' (when' (At t3) (scaleK' 500 GBP)) (and' (when' (At t1) (scaleK' 50 GBP)) (when' (At t2) (scaleK' 50 GBP))`

Findel `And(At(t3, Scale(500,One(GBP))), And(At(t1, Scale(50, GBP)), At(t2, Scale(50, GBP))))`

Future
The counterparty and holder agree to execute contract `c` at time `t`.

Merchant `when' (At t) c`

Findel `Timebound(t -` δ`, t +` δ`, c)`

European option
The holder can choose at `t` whether to acquire contract `c`.

Merchant `when' (At t) (or' c zero')`

Findel `Timebound(t -` δ`, t +` δ`, Or(c, Zero))`

American option
The holder can choose until time `t` whether to acquire contract `c`.

Merchant `anytimeO' (Before t) c`

Findel `Timebound(now, t +` δ`, Or(c, Zero))`

Binary option
The holder receives 100 GBP if an event occurred (i.e. a boolean observable/gateway at `addr` reports *true*)

Merchant `cond' (External addr) (scaleK' 100 (one' GBP))`

Findel `If(addr, Scale(100, One(GBP)), Zero)`