SCHOOL OF COMPUTER SCIENCE

Formalisation of a Financial Combinator Library in Agda

Deepkumar Pawar

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.

Wednesday 4th September, 2024

Abstract

I present a short introduction about dependent types, the dependently typed programming language Agda and how it can be used to prove theorems.

I then present a case study into how Agda can be used to write a declarative combinator library and prove theorems and properties using the definitions of the encoded combinators. I build upon the work conducted by Jones and Eber[6][7]. They use tools from functional programming and formal semantics to describe and value financial contracts. Specifically, they introduce a combinator library in Haskell to describe contracts and a compositional denotational semantics to compute what a contract is worth.

In this paper, I showcase how a combinator library can be implemented in a dependently typed theorem proving language such as Agda and how properties and theorems can be proved using it.

Dedication and Acknowledgements

I thank my supervisors, Steven Ramsay, Jess Foster and Theodoros Constantinides for all of their support and guidance.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others including AI methods, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Deepkumar Pawar, Wednesday 4th September, 2024

Contents

List of Figures

Ethics Statement

This project did not require ethical review, as determined by my supervisor, Steven Ramsay.

Chapter 1

Introduction

Billions of pounds worth of assets get traded every day. Assets include stocks, bonds. currencies commodities and more. At some point, we discovered value and profit in the subtle agreements or contracts that govern when and under what conditions we trade these assets which led to the trade of these contracts themselves between the various entities that possess an interest. This is a financial contract, an agreement between two parties that enforces financial rights and obligations on them.

The reader would note that in today's world, dominated by computers, we have devised use cases for automated computation in the majority of profit making industries, and so the exchange platforms for these financial contracts are no different. It is important and inherently lucrative to be able to automatically and quickly value a contract using its underlying asset to allow the bearer to make informed decisions on its trade.

Consider the following example of a contract *C*.

C: Both of:
 (1) Receive £100 on the 1st of January, 2025
 (2) A choice to be made on the 1st of January, 2025 between:
  (i) Pay 125% of the market rate of 50 shares of 'X' on the 1st of June, 2025
  (ii) Pay £50 on the 1st of June, 2025

Notice how this contract is compositional. It is the combination of clauses *(1)* and *(2)* which could be treated separately as contracts in their own right. Furthermore, *(2)* itself is composed of *(i)* and *(ii)* which could be treated as contracts in their own right as well, that are combined but in a different manner. This was the key observation that inspired the papers by Jones and Eber[6] [7](we refer to it in this paper as the "composing contracts paper"). Based on this observation, they were able to define a set of combinators in Haskell that can be used to describe a wide variety of contracts as well as value (valuation) semantics that allow us to determine the value of a contract. The fundamental idea behind these valuation semantics is that they are compositional which means that the value of a contract is determined by the values of the sub contracts that it is composed of. Finally, they sketch an implementation of their valuation semantics, but note that their first two contributions will receive biased attention in this paper. This work has produced a solution to tame the vast set of financial contracts traded today and provide a way of valuing them. But the question remains, are these arbitrarily chosen combinators and defined valuation semantics *correct*? Do they obey the laws and rules that we would expect them to behave? The original paper presents examples of contracts defined using these combinators but how are we confident that a different combination of these combinators to describe a contract is valued accurately? Can a contract be described in two different ways, and if so, are they valued the same?

For example, in the contract shown above, one might note that we could switch the order in which *(1)* and *(2)* are written and write a new contract that looks slightly different. Intuitively however, the value of the contract must remain unchanged. Therefore, are these two contracts are valued the same and can we *prove* it irrespective of the constituent contracts?

The original contracts paper [6] makes the claim that their defined combinator library satisfies a rich set of equalities and due to the simplistic nature of the valuation semantics, they give rise to simple algebraic properties. In this paper, we explore this claim by (a) demonstrating how proofs of propositions such as the one above are derived from the proofs of algebraic properties that are already known and, (b) assessing the extent to which the dependently typed programming language Agda is a convenient tool to formalise and check such proofs.

These are some of the questions we answer in this paper and given developed ideas of the combinator library, we turn towards formal proofs.

A formal proof is a finite sequence of sentences, each of which is an axiom, an assumption, or follows from the preceding sentences in the sequence by a rule of inference. It differs from a natural language argument in that it is rigorous, unambiguous and mechanically verifiable[3]. Due to this nature, we can encode a traditional proof in the language of a formal proof by defining relevant axioms, assumptions and theorems precisely and use a computer to verify that the proof is correct. Multiple languages exist for this purpose and are usually referred to as 'theorem provers' or 'proof checkers'. Their mechanisms derive from the Curry-Howard Correspondence that states that there is a direct relationship between programs and proofs. Concisely, we are able to encode propositions in the type of a term and by showing that the type is inhabited, that is the term exists and can be defined, we have effectively given a proof for the proposition itself.

The language used in this paper is a dependently typed language called Agda. Agda allows us to use its powerful type system to define terms and combinators as well as state and prove theorems using those combinators. Its type system is more expressive than Haskell in that it lets you define types such *Vector Integer 5* which is the type of a vector of integers with a fixed length of five. This is what is known as a dependent type because the type depends on the value *5*.

The following is an example of a program in Agda that represents a proof that the reader would be able to make sense of by the end of this paper. It is a proof that addition is associative.

```
+-assoc : ∀ (m n p : Nat) → (m + n) + p ≡ m + (n + p)
+-assoc zero n p =
  begin
    (zero + n) + p
  ≡⟨⟩
    n + p
  ≡⟨⟩
    zero + (n + p)
  ∎
+-assoc (suc m) n p =
  begin
    (suc m + n) + p
  ≡⟨⟩
    suc (m + n) + p
  ≡⟨⟩
    suc ((m + n) + p)
  ≡⟨ cong suc (+-assoc m n p) ⟩
    suc (m + (n + p))
  ≡⟨⟩
    suc m + (n + p)
  ∎
```

For now, we only draw your attention to the type signature of the term declared in the first line which states the claim to be proved and the two distinct definitions of that term which are what constitute the proof itself. To check the proof, we simply run it by Agda's type checker which will ensure the definitions return the right type, i.e. that the proof is correct. In chapter 2, we discuss Agda in great detail and equip the reader with all the necessary tools to follow along with the contents of chapters 3 and 4.

The key contributions of this paper are:

- An accessible introduction to Agda including a discussion on dependent types, defining terms and combinators, defining functions and theorem proving
- An exposition of the combinator library and valuation semantics defined in the composing contracts paper

- A comprehensive and concrete implementation of the combinators as well as their valuation semantics in Agda
- A demonstration and examples on how our Agda implementation allows the user to write proofs of equalities that the combinators satisfy
- Support for the claim that the simple semantics by design give rise to simple algebraic properties

Chapter 2

Background

We divide this chapter into two sections. In this first, we present an accessible introduction to the reader on dependent types, Agda and theorem proving. In the second, we provide a simple introduction to financial contracts and the related terms that might be used in this paper.

2.1 Introduction to Agda

This introduction uses elements from *Programming Language Foundations in Agda*[8], *Dependently Typed Programming in Agda*[5] and the Agda Documentation[1].

2.1.1 Dependent Types

Type systems are traditionally used to catch run-time errors at compile-time. Suppose we defined a function in Haskell that took two vectors of the same length and added them together element-wise.

```
vectorAdd :: [Int] -> [Int] -> [Int]
vectorAdd [x] [y] = [x+y]
vectorAdd (x : xs) (y : ys) = (x+y : vectorAdd xs ys)
```

By declaring the type of `vectorAdd` as shown above, we ensure that the function can only be supplied with vectors containing integers. However, there is nothing stopping us from supplying it with two vectors of different length. In fact, if we try doing so, the Haskell compiler will politely let us know by throwing an exception as the current definition is inadequate for handling vectors of different length. We do not want to amend the definition as that goes against our objective, so let us amend the *type* of `vectorAdd` to constrain the length of the vectors it takes.

Define a type *Vector Integer n* which is the type of a vector of integers of length n where `n :: Nat`, a natural number. Now we may write `vectorAdd :: Vector Integer n -> Vector Integer n -> Vector Integer n` and if we assume our type works as defined, this ensures that `vectorAdd` only takes vectors of the same length.

If we take a closer look at the type *Vector Integer n*, we notice that it depends on a value n. Note that n is variable and might be unknown at compile-time. This dependency on a value of some type where the value is arbitrary and need not be known at compile-time is what defines it as a *dependent type*[5]. Dependent types allow us to write define more expressive types that add another layer of complexity to our types and therefore, another layer of sophistication to catching run-time errors at compile-time. However, this is not their only use as we see in later subsections.

Unfortunately, as the reader is likely aware of, Haskell's type system does not natively support such types (although some packages exist that define fixed length vectors). This is where we turn to *Agda*, a dependently typed programming language based on Martin Löf's *Intuitionistic Type Theory*[4].

In the following subsections, we introduce some of the basic concepts to get started with Agda and understand the rest of the paper.

2.1.2 Datatypes and Functions

We declare new datatypes in Agda using the `data` keyword.

```
data Bool : Set where
true : Bool
false : Bool
```

Here, `Set` is the type of `Bool` and can be colloquially thought of as the type of types you might already be familiar with. More accurately, it is part of a hierarchy of increasingly *larger* types that we shall discuss briefly in a later section, but for now, the intuition that it is the type of types is enough. Since *true* or *false* are the only terms that can be *Bool*, their definition as shown completes the definition of `Bool`.

Let us define the function `not` to toggle a `Bool`.

```
not : Bool → Bool
not true = false
not false = true
```

In Agda, functions are not allowed to crash and so the function definition must cover all possible cases. If that is not the case, the type checker will raise an error and ask you to cover the missing cases. Furthermore, while other languages can infer the type signatures of their functions from their definition, with the introduction of dependent types, this is not always possible so Agda forces us to write the type signature of our functions.

Let us define another datatype, one that would represent the natural numbers.

```
data Nat : Set where
zero : Nat
suc : Nat → Nat
```

This definition is based on the Peano axioms[2]. We define `zero` to be a natural number or `Nat` and use `suc` as a term that takes a `Nat` to return a new `Nat`. Thus, the terms of type `Nat` are `zero`, `suc zero`, `suc (suc zero)` and so on. We can define an isomorphism between the set of natural numbers as they largely known and our terms of type `Nat` to get $0 =$ `zero`, $1 =$ `suc zero`, $2 =$ `suc (suc zero)` and so on and *voilà!* We have our natural numbers.

A question that might arise to the reader is why we implement `suc` in the definition of Nat rather than as a function due to the nature of its type definition. We offer two answers. Firstly, there is no way to provide a definition for such a function further than its type signature. If `n : Nat` then the term `suc` is of type `Nat → Nat` intrinsically and should not require a definition or justification. Secondly, when defining functions on `Nat`, we want the type checker to make sure that it consider `suc n` to be a valid case to ensure we cannot have functions that are defined only on `zero` and not for `suc n`.

We can then define addition as follows.

```
_+_ : Nat → Nat → Nat
zero + m = m
suc n + m = suc (n + m)
```

This is our first use of misfix operators. Agda allows flexibility with misfix operators by using underscores. Simply type an underscore (without white-space) where you want the operands to go and Agda will know how to interpret the operation. At the same time, we can use the same operator in an infix manner by keeping the underscores. This means that `zero + zero` is equivalent to `_+_ zero zero`.

In the addition function defined above we use `n` and `m` as variables of type `Nat` on the left hand side to define the operation on the right hand side. Sometimes, if our function definition does not need the variable such as `n` on the left hand side, we can write an underscore in its place as shown below.

```
giveZero : Nat → Nat
giveZero _ = zero
```

Regarding a logical explanation for the definition of `_+_` above, we leave that as an exercise to the reader.

2.1.3 Dependent Datatypes and Functions

Consider a function that takes a type and an element of that type and returns the same element and call it `identity`. This is our first example as we gradually introduce dependent function as it takes in a variable type (which acts as a *value* of `Set`) and a term of that type. While this is not quite a true dependent type as is commonly known, it can be considered as one by definition. Usually, we reserve the name "dependent type" to mean types that depend on a value of a type rather than a value which is some type itself.

In Agda we would write this function as follows.

```
identity : (A : Set) → A → A
identity _ x = x
```

The `(A : Set)` part of the type signature is what defines the dependency on `A`. In this example, we have to explicitly write the type `A` when using `id`. So `identity Nat zero` would evaluate to `zero`. But we do have to explicitly mention the type `A` to use such a function. Consider the alternate function definition as follows.

```
id : {A : Set} → A → A
id x = x
```

Here we make use of what is known as an *implicit argument*. The `{A : Set}` acts the same way as `(A: Set)` except when using the function, the type `A` is inferred by the Agda type checker and so we do not need to provide it.

Sometimes, we might still wish to make use of the implicit argument in the definition of our function. We can do so as follows.

```
getType : {A : Set} → A → Set
getType {A} x = A
```

Consider another example for an if-then-else style function.

```
if_then_else_ : {A : Set} → Bool → A → A → A
if true then x else y = x
if false then x else y = y
```

Here, similar to above, we make use of `A` to write a dependent type whose value might be unknown at compile-time. This is also another example of the misfix mechanism in Agda that allows us place underscores in arbitrary places.

We shall now show how to define a list in Agda. This would correspond to the type `[a]` in Haskell.

```
data List (A : Set) : Set where
[] : List A
_::_ : A → List A → List A
```

While this can be thought of as a dependent type, we specify that `List` is a parametric type parameterised by `A`. It ensures that whenever we declare a List, we must also specify the type of the elements in that list. The empty list `[]` is of type `List A` and we also specify the term `_::_` that lets us append elements to a list thereby returning a new `List A`, analogous to `suc` for `Nat`. Therefore, `x :: xs` is similar to `(x : xs)` in Haskell where `x : A` and `xs : List A`.

We are finally in a position to define a Vector of a specified type and a fixed length, similar to the one discussed at the beginning of this section.

We define it as follows.

```
data Vec (A : Set) : Nat → Set where
[] : Vec A zero
_::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

Immediately, we draw similarities to the definition of `List`. However, the key distinction is the type signature where we supply an extra term of type `Nat` to define the length of our vector. Therefore, we are able to define the type `Vector Integer n` for some `n : Nat` as discussed at the beginning of the section. Note the usage of the implicit argument for `_::_` which works in the expected way. This is our first example of a true dependent type as they are commonly known.

We can now define `vectorAdd` and we do so as follows.

```
vectorAdd : {n : Nat} → Vec Nat n → Vec Nat n → Vec Nat n
vectorAdd [] [] = []
vectorAdd (x :: xs) (y :: ys) = (x + y) :: (vectorAdd xs ys)
```

Here, the `{n : Nat}` part of the type signature constrains the length of the two vectors to be the same. A consequence of this are the cases for pattern matching here. We do not require a case for `vectorAdd [] (y :: ys)` because they have different length and therefore would not match the function type. Another example is as follows.

```
head : {A : Set} {n : ℕ} → Vec A (suc n) → A
head (x :: xs) = x
```

Here, this function that returns the head of vector does not need a case for `[]`. In fact, any attempts to add one would be met by an error raised by the type checker. We let the reader deduce why this is the case (*hint*: check the type signatures of `head` and `[]`).

2.1.4 Universe Levels

Russell's paradox states that the set of all sets cannot be a set. This idea is encapsulated in Agda by the fact that the type of a `Set` cannot be a `Set`. However, sometimes it is useful to be able to work with the type of a `Set` which is why we use a different type called $\texttt{Set}_1$ so that $\texttt{Set : Set}_1$. However, by similar logic, we cannot have $\texttt{Set}_1\ \texttt{:}\ \texttt{Set}_1$ so Agda defines a new type $\texttt{Set}_2$ so that $\texttt{Set}_1\texttt{:}\ \texttt{Set}_2$. This continues indefinitely and so we have `Set`, $\texttt{Set}_1$, $\texttt{Set}_2$, $\texttt{Set}_3$ and so on. In fact, `Set` is just short for $\texttt{Set}_0$. In many ways, we find that terms of type $\texttt{Set}_1$ behave similarly to those of type `Set`. However, we find that elements of type $\texttt{Set}_1$ are potentially larger those of `Set` and so sometimes we say that $\texttt{Set}_1$ is a large set.

A type whose elements are types is known as a *universe*.

2.1.5 Libraries and Imports

Many of the datatypes, functions or proofs you might want to use have already been defined in a different module, sometimes in the Agda standard library. To use them we make use of the keywords `import`, `open` and `using`. As an example of this, let us import the definitions for `Nat` that we manually defined in a previous section. We use the Agda standard library to find the `Data.Nat` module that contains the relevant definitions.

```
open Data.Nat
import Data.Nat using (ℕ; zero; suc; _+_; _*_)
```

The first line brings the module into scope. The second line opens that module and brings all the names mentioned inside the parentheses into the current scope. We separate the names with a semicolon as that is one of the few characters that are not allowed to be part of an Agda name. Note that in this library, `Nat` is named ℕ.

We can also combine both these lines into one as follows.

```
open import Data.Nat using (ℕ; zero; suc; _+_; _*_)
```

2.1.6 Proofs and Reasoning

In this section, we demonstrate how proofs in Agda are written. The ideas used are based on the Curry-Howard Correspondence where we derive a relationship between programs and proofs. We start by immediately importing the following from the Agda standard library.

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; _∎)
```

In the first line we bring the named module into the current scope and assign it the name `Eq`. In the second and third lines we open the module and another module inside it and bring the terms mentioned in the parentheses into the current scope.

The term `_≡_` is the equality operator and `refl` is a term that provides *evidence* that two terms are equal. We shall discuss what we mean by evidence shortly. Also, we shall take the terms `cong`, `sym`, `begin_`, `_≡⟨⟩_` and `_∎` as given and demonstrate their use below as they come up.

As a first example, let us prove that `1 + 0 = 1`. We do this by declaring a term with a type signature as follows.

```
addOneZero : (suc zero) + zero ≡ (suc zero)
```

To prove this claim, we simply need to write the definition of `addOneZero`. By doing so, we would have provided a term of type `(suc zero) + zero ≡ (suc zero)` which then serves as *evidence* for the corresponding proposition we wish to prove. We use the operators imported above to do this.

Here is where we note that it is important to know and understand the definitions of the imported terms to fully understand how we obtain the right term that serves as evidence for the type signature of `addOneToZero`. However, we refrain from going into those details here and rather prioritise being able to apply these terms to write these proofs as that is enough for our purposes, even later in this paper. We shall build intuition for effective application by making analogies to writing a traditional proof.

We write the following.

```
addOneZero : (suc zero) + zero ≡ (suc zero)
addOneZero =
  begin
    (suc zero) + zero
  ≡⟨⟩
    suc (zero + zero)
  ≡⟨⟩
    suc zero
  ∎
```

We start by using the `begin` operator which acts as a signal to the start of our proof. We write the left hand side and then use the ≡⟨⟩ operator in a similar way to how one might write an equals sign in a traditional proof to signal a simplification (or expansion) of our terms. Here, we simply apply the definition of addition which is the same as defined in an earlier section. We apply it again to obtain `suc zero`, which corresponds to the right hand side of our claim. We use the ∎ operator to signal the end of our proof and compile our program. If the Agda type checker finds no problems with what we have written, we have successfully proved our claim (and have supplied `addOneZero` with sufficient evidence with respect to its type signature).

Note that in this proof, we have merely applied the definition of the addition operator. Agda's type checker is smart enough to do this automatically and so it already knows that `(suc zero) + zero ≡ (suc zero)`, which is why are able to write the following proof instead.

```
addOneZero : (suc zero) + zero ≡ (suc zero)
addOneZero = refl
```

The term `refl` where "refl" stands for reflexive is *evidence* that a term is equal to itself. As mentioned, Agda knows that `(suc zero) + zero ≡ (suc zero)` so to provide evidence of something Agda can evaluate itself, we need only supply it with the term `refl`.

For the remainder of this section, we present a proof for the commutativity of addition, that is, the order of operands does not matter.

```
m + n = n + m
```

For this proof, we make use of two lemmas. The first lemma states that zero is what is known as right-identity which means that adding zero to the right of natural number gives us the same natural number.

```
m + 0 = m
```

Note that the base case in the definition for `_+_` covers zero as left-identity, `0 + m = m`. We write the statement for this lemma and present a proof as follows.

```
+-identityʳ : ∀ (m : Nat) → m + zero ≡ m
+-identityʳ zero =
  begin
    zero + zero
  ≡⟨⟩
    zero
  ∎
+-identityʳ (suc m) =
  begin
    suc m + zero
  ≡⟨⟩
    suc (m + zero)
  ≡⟨ cong suc (+-identityʳ m) ⟩
    suc m
  ∎
```

This is an inductive proof where we have a base case for `+-identityʳ zero` and an inductive step case `+-identityʳ (suc m)`. In the statement we have written `∀ (m : Nat) → m + zero ≡ m`. The ∀ symbol stands for "for all" and can be put for clarity, but omitting it would not produce any errors. The statement depends on `m : Nat` as the placeholder for any natural number for which this lemma should hold true for. This also means that when writing the proof, we must provide a term of type `Nat` to `+-identityʳ`.

The base case uses `zero` and the evaluation is trivial. The inductive step case uses `suc m` and starts by applying the definition of `_+_`. We then write `≡⟨ cong suc (+-identityʳ m) ⟩` which functions similarly to `≡⟨⟩` but provides additional evidence or justification for making that forward step. This is also the first time we mention `cong` which is short for *congruent.* We wish to make the step from `suc (m + zero)` to `suc m`. In an inductive argument, (informally) we assume that the proposition that we are trying to prove for a "k+1" term holds for "k". Similar to that we use `+-identityʳ m` as justification that the lemma holds for `m` as we evaluate the case for `suc m`. So we know that `m + zero` is `m`. However, we have to step from `suc (m + zero)` so we apply `suc` to both sides of `+-identityʳ m` and justify the *congruence* of doing so by using the term `cong` as shown. We are left with `suc m` and the conclusion of our proof follows.

Commenting additionally on the inductive step, when applying `+-identityʳ m`, note that the Agda type checker uses recursion to ensure that `+-identityʳ m` is truly justified. With each recursive step our `m` gets smaller until it becomes `zero`, where the base case we proved comes into play and justifies the `+-identityʳ m` as a consequence.

The second lemma we shall prove pushes `suc` on the second operand to the outside.

```
m + suc n ≡ suc (m + n)
```

Note that the inductive case in the definition for `_+_` does the same for the first argument.

We present the statement and the proof of this lemma as follows.

```
+-suc : ∀ (m n : Nat) → m + suc n ≡ suc (m + n)
+-suc zero n =
  begin
    zero + suc n
  ≡⟨⟩
    suc n
  ≡⟨⟩
    suc (zero + n)
  ∎
+-suc (suc m) n =
  begin
    suc m + suc n
  ≡⟨⟩
    suc (m + suc n)
  ≡⟨ cong suc (+-suc m n) ⟩
    suc (suc (m + n))
  ≡⟨⟩
    suc (suc m + n)
  ∎
```

The proof for this lemma is inductive and quite similar to the first lemma. This time however, we have two dependent type definitions in the proposition `∀ (m n : Nat) → m + suc n ≡ suc (m + n)`. They function as expected wherein we now have to provide two arguments to `+-suc` and for this proof, we write `n` as a general natural number for the second argument and apply induction on the first.

Finally, we can now define and prove commutativity as follows.

```
+-comm : ∀ (m n : Nat) → m + n ≡ n + m
+-comm m zero =
  begin
    m + zero
  ≡⟨ +-identityʳ m ⟩
    m
  ≡⟨⟩
    zero + m
  ∎
+-comm m (suc n) =
  begin
    m + suc n
  ≡⟨ +-suc m n ⟩
    suc (m + n)
  ≡⟨ cong suc (+-comm m n) ⟩
    suc (n + m)
  ≡⟨⟩
    suc n + m
  ∎
```

Again, this follows from the same principles and ideas of induction as shown earlier except we apply induction on the second argument rather than the first, which we keep general. In different places we use the terms `+-suc` and `+-identityʳ` to apply the lemmas that justify their respective steps.

The proof for commutativity was chosen as an example because we use this property later on in chapter 3 to justify a step in a different proof. With this coverage, the reader is equipped with the understanding required to make sense of the proofs that we will explore further in the paper.

2.1.7 Postulates

A postulate is a declaration of an element of some type without an accompanying definition. It allows us to write propositions without needing proof. For example, we may write the following.

```
postulate
  +-assoc : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
```

We are now free to use this property without needing to provide any further proof (in the form of a definition of `+-assoc`).

Postulates are typically used to write a proposition which we have already proved elsewhere and know to be true but consider its proof as tedium. They can also be used to write *axioms* as those statements that are self-evident and therefore require no proof to be true.

Naturally, one must take great care when using postulates as asserting a false identity would lead to a breakdown in our logical system.

2.1.8 Semantics

We now take our focus to a more general setting away from Agda to briefly discuss what are known as *semantics*.

In declarative or functional languages, we might write a set of combinators designed to tackle a domain-specific task. We sometimes call them *domain specific languages*. Formally, they are languages written at a higher level of abstraction optimised for a specific class of problems.

In some cases, to make sense of these combinators, a solution is to translate them to *mathematical objects* that we are more familiar with. By *mathematical object*, we loosely mean any entity that is bound by certain defined rules and logic and are by nature easier to understand or interpret. By *make sense*, we mean to derive some sort of meaning from these combinator terms to an extent fit for our purposes. Sometimes, we might simply define this translation because the image of this translation has operators that are already defined to save ourselves time and effort. The function we use for this translation is called its semantics or a semantics library. The vagueness of the language used to describe these is intentional as it is highly dependent on the specific purpose and needs. So let us treat this with a simple example.

Suppose we write a definition for strings that looks like the following in Agda.

```
data String : Set where
  empty : String
  str₁ : String
  str₂ : String
  append : String → String → String
```

We have an empty string, two different strings and a combinator that appends two strings. In our example, let us be concerned with a *weighted length* of these strings where `str₁` is worth *1* unit and `str₂` is worth *2* units. We define the following semantics function `eval`.

```
eval : String → Nat
eval empty = zero
eval str₁ = suc zero
eval str₂ = suc (suc zero)
eval (append u v) = (eval u) + (eval v)
```

Note that it takes a string and returns a natural number. We would say here that we are defining a *natural number semantics* for `String`. Similar to any other function in Agda, it must have cases for all possible `String` terms. We trust that its definition is clear to the reader, but we still bring our focus to the definition of `eval (append u v)`. Note that the definition is recursive over the constituent strings `u` and `v` that make up the string `append u v` combined by addition. This is where we say that our combinators and semantics are *compositional* in nature as strings can be composed of other strings and the semantics can decompose them as necessary.

The most important feature of this semantics function is that it results in a natural number. Since we are already quite familiar with natural numbers and likely have various operations defined on them, we can apply them and *make sense* of them as needed.

2.2 Financial Contracts

In this section, we tour some of the fundamental and background knowledge required to understand the execution part in chapter 3 of this paper. All the type definitions and semantics are taken from the composing contracts paper.

2.2.1 Initial Concepts

The purpose of this section is not to equip the reader with strongly worded definitions on financial contracts and derivatives as they are used in the real world. Instead, we assume no background knowledge and aim to build a rough intuition to help understand the combinator library defined later.

A contract refers to an agreement between two parties to exchange currency at a point in time, potentially subject to conditions. For example, consider the agreement between a parent and their child wherein the parent promises the child a gift in the form of a payment of £100 on the child's 16th birthday. Whilst not legally enforceable (in the likely scenario), we may define this as a contract as there are two parties, an established amount to exchange and a defined date when the exchange must take place.

In some cases, these contracts might depend on some set conditions. Suppose in the example above, the parent agrees to gift their child but only if they score well enough on a school test. This condition introduces a new variable into the mix, the child's test score, and so on their birthday we *observe* the test score and if it is above a certain threshold, only then is the parent *obligated* to pay the agreed amount.

Whilst being rather contrived, this example demonstrated how a contract might depend on an external variable, the child's test score. Instead, suppose the following. Parties A and B typically trade oranges where A buys from B during the harvest season. However, due to uncertainties in demand, B wishes to be able to guarantee sales the following year. Therefore, they come up with an agreement with A to have the choice to sell them a specified number of commodities (the oranges) the following year. In return, they offer a discounted rate to A. The key note here is that B possesses an choice, not an obligation, which means that B can choose not to sell to A at all the following year, should they wish. This type of financial contract is known as an *option*.

Finally, let us describe what it means for a contract to have a *value* and how that might change. Consider the example we just discussed. There is no reason to believe that A is the only buyer of oranges in the market and hence let C be such a buyer. As the contract in question has the possibility to offer A a discounted rate, it is a desirable contract to possess and thus has value. Therefore, A can sell the contract itself to another buyer such as C for a price. This will then bestow C with the rights of the contract, meaning that B will have the choice to sell their oranges to C rather than A. It is important to note that this price or value of this contract might fluctuate in relation to various factors such as respective demands, the market rate of oranges, the presence of other sellers, and so as time goes along. This sets us up nicely to talk about the *acquisition date* of a contract, which is the date a contract is acquired by a party whereby the rights and obligations of it take effect. The value of a contract will likely depend on its acquisition date. For example, if the contract held by A is acquired by C two years from now, we would notice that all the rights and obligations which were supposed to be fulfilled a year from now would be a year old at that point. Therefore, that contract would be worthless as its benefits have expired. The acquisition date of a contract is therefore crucial to defining a contract.

2.2.2 Justifying a Combinator Library

Now we have built the intuition necessary, we proceed to justify a combinator library.

Let us continue to use the example described above, where B has the option to, in 1 year's time, sell their (let us say, 50) oranges to A at a discounted price *or* do nothing.

Firstly, observe that we have actually defined two contracts in this example, they are separated by a disjunction. The first contract is the obligation to sell 50 oranges to A at a discounted price and the second is a contract that does nothing, that is, has no rights or obligations at all. Let us give these contracts a name, the first we shall call *discountedSale*, the second we shall call *zeroContract* and the whole disjunction we shall call *yearOption*. As they are all contracts, let us treat them as terms and assign them a type, `Contract`, as follows.

```
yearOption : Contract
discountedSale : Contract
zeroContract : Contract
```

Succinctly, we say that `yearOption` is *composed* of `discountedSale` and `zeroContract`, using a disjunction. However, notice that this disjunction is independent of the exact details surrounding the contracts `discountedSale` and `zeroContract`. In fact, we could substitute these two contracts with any other contract and still be able to choose between them. Therefore, let us abstract this functionality out as its own combinator and call it `or`. Clearly, it takes in two contracts to give us a third and so we write the following.

```
or : Contract → Contract → Contract
```

Thus we can write `or discountedSale zeroContract` to describe the contract that arises from having a choice between `discountedSale` and `zeroContract`. We may also write `discountedSale 'or' zeroContract` where `'or'` is the infix version of `or`.

However, we cannot yet equate this to `yearOption` because if `discountedSale 'or' zeroContract` is acquired at the current date, it forces the bearer to make the choice immediately whereas `yearOption` should let the bearer make the choice in a year's time. To model this, we use two new combinators, `when` and `at`. We define them as follows.

```
when : Boolean → Contract → Contract
at : Date → Bool
```

The contract `when o c`, where `o` is a `Boolean` and `c` is a `Contract`, grants the bearer the same rights and obligations as `c` but only when `o` evaluates to *True*. The term at `d` where `d` is a `Date`, evaluates to True only when the current date is `d`.

Let us call the date a year from now, `d`. Consider the contract `when (at d) (discountedSale 'or' zeroContract)`. This contract bestows the bearer with the rights and obligations of `discountedSale 'or' zeroContract` only when `at d` evaluates to True, which is on the date `d`, and is useless on any other date. Thus, this is now an accurate equivalence for `yearOption` and we can write following.

```
yearOption = when (at d) (discountedSale 'or' zeroContract)
```

Let us delve deeper into `discountedSale`. It is the contract that obligates the bearer to sell 50 oranges to the counter-party (the other party). Suppose the discount is of 10%. The first aspect to note is that it is a contract that obligates the bearer to perform a sale. We assume that a contract natively refers to receiving in an exchange, so let us define a combinator that represents the opposite of receiving in the form of a sale.

```
give : Contract → Contract
```

Next, we want to define a contract for the discounted purchase (opposite of sale before applying give) of 50 oranges so let us model a floating point variable called `orangePrice` that tells us the price of an orange at any given point of time with respect to a currency. We would refer to this kind of variable as an *observable* as defined in the next section but we forgo that for now. Now we want to scale `orangePrice` by 50, apply the discount and then multiply that by the value of 1 unit of that specific currency to obtain the desired contract. For this, we define the following combinators.

```
scale : Double → Contract → Contract
one : Currency → Contract
```

The combinator `scale` takes a double value and a contract and scales the value of the contract by that double value. The combinator `one` takes a currency and returns a contract that gives the bearer the right to receive exactly one unit of that currency on the date of acquisition.

Therefore, we write the desired contract as `scale (orangePrice * 50 * 0.9) (one GBP)` to represent a contract that gives the bearer the right to be paid of the cost of 50 oranges discounted by 10% in pounds. So if the price of an orange was £1, the bearer of the contract would be paid £45. As a whole we can use the `give` combinator to write `discountedPrice = give (scale (orangePrice * 50 * 0.9) (one GBP))` and therefore the following.

```
yearOption = when (at d) (give (scale (orangePrice * 50 * 0.9) (one GBP)) 'or'
             zeroContract)
```

The combinators defined here are based on the ones we discuss in the upcoming sections. As a final example, let us use the one first mentioned in the introduction.

C: Both of:
  (1) Receive £100 on the 1st of January, 2025
  (2) A choice to be made on the 1st of January, 2025 between:
    (i) Pay 125% of the market rate of 50 shares of 'X' on the 1st of June, 2025
    (ii) Pay £50 on the 1st of June, 2025

Let `d1 d2 : Date` and the variable that models the market rate of 1 share of 'X' be `shareValue`.

```
d1 = 01/01/25
d2 = 01/06/25
```

We propose the following contract definition.

```
C = (when (at d1) (scale 100 (one GBP))) 'and'
    (when (at d1) ((when (at d2) (give (scale (shareValue * 50 * 1.25) (one GBP)))) 'or'
                   (when (at d2) (scale 50 (one GBP)))))
```

This is quite a dense definition so let us break it down by considering the definitions of (i) and (ii) first.

```
(i) = (when (at d2) (give (scale (shareValue * 50 * 1.25) (one GBP))))
(ii) = (when (at d2) (scale 50 (one GBP)))
```

These definitions are simple and are quite similar to the example discussed earlier. We then combine these two contracts into an option that is exercisable on the date `d1`.

```
(2) = when (at d1) ((i) 'or' (ii))
```

Then we write the definition for (1).

```
(1) = (when (at d1) (scale 100 (one GBP)))
```

Finally, we combine (1) and (2) using `and`.

```
C = (1) 'and' (2)
```

Expanding each definition would explain the full contract definition for C we wrote earlier.

2.2.3 Observables

In these examples, we mention multiple external factors that we use to value a contract or the subcontracts that compose it. These include the market rate for oranges or even the test scores of the student in the very first example. They act as variables that take values as a function of time. In the composing contracts paper, an *observable* is defined as a time-varying, maybe unknown in advance, term.

When defining an observable, we say that the observable term is of type `Obs a`, where a is the type of the values that the observable term can take.

For example, if we want to declare an observable term to model the market rate of oranges we say the following.

```
marketRate : Obs Double
```

This is because we expect the specific values that `marketRate` would take to be of type `Double`.

In a different example, we may wish to model whether the strength of the euro exceeds that of the dollar and would therefore write `euroStrongerThanDollar : Obs Bool` as the value of this observable would be either True or False.

The key intuition here is that we have a quantity that might fluctuate based on time. When supplied with a specific time, usually in the future, the observable tells us the possible values of that quantity. Note that we say *possible values* rather than the exact value. This is because in most case, it is impossible to predict the future value of a quantity with complete certainty. Thus, instead, we obtain a set of possible values with associated probabilities, not unlike a *random variable*. We reiterate this point and build on it later in this section.

Finally, note that we define combinators related to observables which are shown in the next subsection alongside a brief description.

2.2.4 Contracts and Observables Combinator Library

Now we are in a position to showcase the definitions of the combinator library. See Figures 2.1 and 2.2 where we use the original definitions taken directly from the composing contracts paper and show them here for reference. They define them in Haskell but we convert them to Agda later in chapter 3.

2.2.5 Valuation Semantics and Value Processes

We have defined the combinators used to build financial contracts and observables. Although we colloquially understand what a contract means, we will now define a semantics library to give us a precise way to value these contracts.

The composing contracts paper defines two separate "layers" to express contract valuation, the Abstract Valuation Semantics (AVS) and the Concrete Implementation (CI). The AVS translates combinators into a *value process* equipped with mathematical operations that describe the combinators for contracts defined in the previous subsection. These factor in uncertainty associated with valuing a contract for a future point in time and are discussed below. The CI is dependent on the *financial model* that directly implements the uncertainties using some discrete *numerical method*.

In this paper, we shall concern ourselves only with the AVS as the CI goes beyond the scope of our objectives. However, you can seek further details in the composing contracts paper.

`zero :: Contract`
  `zero` is a contract that has no rights and no obligations

`one :: Currency -> Currency`
  If you acquire `(one k)` you immediately receive one unit of the currency `k`

`give :: Contract -> Contract`
  To acquire `(give c)` is to acquire all `c`'s rights and obligations, and vice versa

`and :: Contract -> Contract -> Contract`
  If you acquire `(c 'and' d)` you immediately acquire both `c` and `d`

`or :: Contract -> Contract -> Contract`
  If you acquire `(c 'or' d)` you must immediately acquire your choice of either `c` or `d` (but not both)

`cond :: Obs Bool -> Contract -> Contract -> Contract`
  If you acquire `(cond b c d)` you acquire `c` f the observable `b` is true at the moment of acquisition, and `d` otherwise

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

`when :: Obs Bool -> Contract -> Contract`
  If you acquire `(when o c)`, you must acquire `c` as soon as observable `o` subsequently becomes True. It is therefore worthless in states where `o` will never again be True.

`anytime :: Obs Bool -> Contract -> Contract`
  Once you acquire `(anytime o c)`, you may acquire `c` at any time the observable `o` is True. The compound contract is therefore worthless in states where `o` will never again be True.

`until :: Obs Bool -> Contract -> Contract`
  Once acquired, `(until o c)` is exactly like `c` except that it must be abandoned when observable `o` becomes True. In states in which `o` is True, the compound contract is therefore worthless, because it must be abandoned immediately.

Figure 2.1: Contracts

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

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

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

`date :: Obs Date`
  The value of the observable `date` at date `s` is just `s`

Figure 2.2: Observables

The semantics library or valuation semantics is a function that takes a term of type `Contract` or `Observable` and gives us a mathematical object that would help tell us what a contract is worth. This allows us to have a precise way of comparing two contracts or determining if they are equal. They also allow us to reason with these contracts as we shall discuss and demonstrate later.

Firstly, we define a *value process* which will serve as the mathematical object mentioned earlier that our valuation semantics will translate contracts and observables into. A value process is a function that takes in a time and gives us a random variable that describes the possible values that the value process might exhibit at that time. They are indexed by a type, similar to an observable and we use `PR` to construct them, similar to how `Obs` constructs an observable.

```
PR a = Date → RV a
```

Thus, if we say `A : PR Double` and `d : Date`, then `A d` will give us a random variable of type `RV Double` which means that the values they take will all be of type `Double`.

As another example, define a value process `ManUtdWinPL : PR Bool` that given a date tells us if the football team Manchester United had won the *premier league* trophy the year of that date. If we define `d1 : Date` such that `d1` is the 30th of May, 2025 (the end of the next season at time of writing), we can use this date to get a random variable `winRV1 : RV Bool` that tells us the probabilities of the value process being either True or False.

```
ManUtdWinPL d1 = winRV1
```

`winRV1` could be expressed as the following, clearly showcasing the probabilities.

```
winRV1 = {
  True - 80%
  False - 20%
}
```

Now define `d2 : Date` where `d2` is the 30th of May, 2023. Note that this is a date in the past and we already know the outcome in this case. Defining `winRV2 : RV Bool` we can then write the following.

```
ManUtdWinPL d2 = winRV2

winRV2 = {
  True = 0%
  False = 100%
}
```

This shows how a determined outcome can expressed using a random variable returned by a value process.

Apart from some of the optimism showed in this example (`winRV1`), it demonstrates how a value process would work in practice.

The valuation semantics for contracts translate contracts into value processes of type $\mathbb{R}$, where $\mathbb{R}$ refers to the real numbers. This is to signify that all contracts have a real-valued valuation. Furthermore, we accompany these values with a currency as that is fundamental to make sense of the valuation. Due to this, we note that the valuation semantics for contracts are accompanied by a placeholder for currency.

For observables, the type of the value process that the valuation semantics translates to is the same as the type of the observable. These are not valuations of contracts and thus are not accompanied by a currency.

Alongside the valuation semantics, we also define some primitives and functions on the value process level that lift familiar mathematical operations to the value process level.

We take the definitions from the original composing contracts paper and present them here as Figures 2.3, 2.4 and 2.5.

Finally, in Figure 2.6, we briefly mention certain properties that we expect to uphold. They arise from how we expect financial models to work.

Unlike the terms and types we have discussed earlier, these are simply equations that we serve as axioms due to the nature of financial systems and so we cannot provide further proof for them. The example we treat here is (A1) as it is the simplest to understand.

```
eval [.] : Currency -> Contract -> PR ℝ

eval k [zero] = K(0)
eval k [one k2] = exch k (k2)
eval k [give c] = - eval k [c]
eval k [o 'scale' c] = val [o] * eval k [c]
eval k [c1 'and' c2] = eval k [c1] + eval k [c2]
eval k [c1 'or' c2] = max(eval k [c1], eval k [c2])
eval k [cond o c1 c2] = if(val [o], eval k [c1], eval k [c2])
eval k [when o c] = disc k (val [o], eval k [c])
eval k [anytime o c] = snell k (val [o], eval k [c])
eval k [until o c] = absorb k (val [o], eval k [c])
```

Figure 2.3: Valuation Semantics for Contracts

```
val [.] :: Obs a -> PR a

val [konst x] = K(x)
val [date] = date
val [lift f o] = lift(f, val [o]
val [ lift2 f o1 o2] = lift2(f, val [o1], val [o2]
```

Figure 2.4: Value Semantics for Observables

`K : a -> PR a`
The process `K(x)` is defined at all times to have the value `x`

`date : PR Date`
The process `date` has as its value the date

`cond : PR Bool -> PR a -> PR a -> PR a`
Conditional choice between the latter two processes, based on the first

`lift : (a -> b) -> PR a -> PR b`
Apply the specified function to the argument process point-wise

`lift2 : (a -> b -> c) -> PR a -> PR b -> PR c`
Combine the two argument processes point-wise with the specified function

`+,*,... : PR ℝ -> PR ℝ -> PR ℝ`
Add, or multiply (etc.) the two processes. Equivalent to `lift2(+)`, etc.

Figure 2.5: Value Process Primitives

```
(A1)  exch k (k) = K(1)
(A2)  exch k2 (k1) * exch k3 (k2) = exch k3 (k1)
(A3)  disc k (K(True), p) = p
(A4)  exch k1 (k2) * disc k2 (o, p) = disc k1 (o, exch k1 (k2) * p
(A5)  disc k (o, p1 + p2) = disc k (o, p1) + disc k (o, p2)
```

Figure 2.6: Properties for Value Processes

(A1) `exch k k = K 1`

This means that the value of any currency `k` expressed in itself returns the constant value process `K 1`. This is justified intuitively as no matter what currency we use, one unit of it will trivially equate to one unit in the same currency regardless of when we decide to observe this relationship.

With this, we conclude the background knowledge required to understand the terminology used in this paper. Following up in chapter 3, we proceed to implement this in Agda.

Chapter 3

Project Execution

3.1 Simplifications

In this implementation of the combinator library and valuation semantics (from chapter 2), we have taken some liberties and simplified certain parts.

- For all cases where the use of real numbers has been mentioned, such as for the valuation of contracts, we have replaced them with integers. This is because of the more intuitive and simpler library support available in Agda for Integers as compared to real numbers. This does not affect our implementation of the combinators or semantics as both integers and reals mostly use similar, if not the same, operators and functions.
- We have simplified the definition of a value process so that instead of `PR a` returning a random variable of type `a`, it will return a value of type `a` itself. This is because we notice that the utility of using a random variable lies only when considering the concrete implementation of these valuation semantics with respect to a financial model that takes into account this stochastic nature. Otherwise, these random variables only add a layer of complexity that is unnecessary for the scope of this paper. Furthermore, all operations we do on values of a type can also be done on random variables of that type and vice versa which means we save on having to define combinators that lift operations between values to operations between those random variables.

3.2 Imports

We begin by importing the following.

```
open import Data.Bool.Base using (Bool; true; false; T; _∧_; _∨_; not; if_then_else_)
open import Data.Integer
open import Data.Integer.Properties
open import Data.Product using (_×_; proj₁; proj₂; _,_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; sym; trans)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; step-≡; _∎)
```

These imports allow us to make use of the following definitions and functionalities:

- Boolean datatype and functions
- Integer datatype and functions
- Product datatype and functions for representing pairs
- Reasoning for writing proofs

3.3 Implementing Date and Currency

While crucial to the model, we present a simple implementation for date and currency.

```
data Date : Set where
  first : Date
  next : Date → Date

data Currency : Set where
  GBP : Currency
  USD : Currency
```

We model `Date` using Peano axioms similar to how natural numbers are (see chapter 2). We do not require any further functionality as this encodes the sequential nature of date and is enough for our purposes.

The `Currency` type serves as a placeholder which is used later in the implementation.

3.4 Implementing Observables and Contracts

We first implement the observables. The type of an observable term depends on a separately provided type by using the constructor `Obs` followed by that type. Therefore, we can assert the type of `Obs` as the following.

```
Obs : Set → Set
```

Recall that we use `Set` to refer to the type of most of the types we use such as Integer, Boolean and such (see chapter 2). However, this does not work. By using the type `Set` to define a value for `Obs` to depend on, we are assigning the term `Set` the type `Set`. This leads to contradictions such as Russell's Paradox and Girard's Paradox. Instead Agda uses a hierarchy of types where `Set : Set₁`. We replace the second `Set` with `Set₁` to obtain the following which works as desired.

```
Obs : Set → Set₁
```

Now we have the type definition out of the way, we can write the following in Agda.

```
data Obs : Set → Set₁ where
  konst : {a : _} → a → Obs a
  lift : ∀ {a b : _} → (a → b) → Obs a → Obs b
  lift₂ : ∀ {a b c : _} → (a → b → c) → Obs a → Obs b → Obs c
  date : Obs Date
```

Under this type we declare `konst : {a : _} → a → Obs a` and note that `konst` takes a value and not a type, unlike `Obs`. This allows us to write `konst 1` to mean the observable of type `Obs Integer` that returns the value one irrespective of what time it is observed at.

`lift` and `lift₂` take in a function from types `a` to `b` and lift that function to the observable level. `date` is defined simply to be the current date of when it is observed and is therefore of type `Obs Date`.

We now implement contracts.

```
data Contract : Set₁ where
  zeroC : Contract
  one : Currency → Contract
  give : Contract → Contract
  and : Contract → Contract → Contract
  or : Contract → Contract → Contract
  cond : Obs Bool → Contract → Contract → Contract
  scale : Obs ℤ → Contract → Contract
  when : Obs Bool → Contract → Contract
  anytime : Obs Bool → Contract → Contract
  until : Obs Bool → Contract → Contract
```

Recall the type definitions and meanings of these combinators from chapter 2. We simply write them in Agda as is.

Notably, we write `Contract : Set₁` for the same reason as we do for `Obs`.

Now we have defined contracts and observables, let us use the examples from chapter 2 and write them in this combinator library.

3.5 Implementing Value Processes

Recall that a value process is a function that is indexed by a type and given a time it returns a random variable of the same type explaining the possible values that that value process could take. Given our simplification, however, the value process returns a single value of the same type rather than a random variable. Therefore, we write the following.

```
PR a = Date → a
```

In Agda, we write as follows.

```
PR : Set → Set
PR a = Date → a
```

We then proceed to define the process primitives as functions in the following way.

```
K : {a : Set} → a → PR a
K x _ = x

datePR : PR Date
datePR d = d

liftPR : {a b : Set} → (a → b) → PR a → PR b
liftPR f x d = f (x d)

liftPR₂ : {a b c : Set} → (a → b → c) → PR a → PR b → PR c
liftPR₂ f x y d = f (x d) (y d)

liftPR₃ : {a b c d : Set} → (a → b → c → d) → PR a → PR b → PR c → PR d
liftPR₃ f x y z d = f (x d) (y d) (z d)
```

The value process constructed using `K` should return the value following the `K` regardless of what date we give it. We represent this using the underscore as a placeholder for date in front of `K x : PR a` where `x : a` and ensure that it evaluates to `x` in all cases.

We refer to the date term on the value process level as `datePR` to avoid confusion.

`liftPR` and `liftPR₂` work similarly where we explicitly supply the date `d` on the left hand side so that we can use it in their respective definitions. Note the new addition of `liftPR₃`. Whilst not being mentioned earlier, we define it in a similar manner to `liftPR` and `liftPR₂` to lift operations that take in three operands.

Next, we implement the model primitives. Recall that they depend on the external factors and operations such as exchange rate and combining that with their primitive nature, we can only postulate them and treat them like axioms. For this, we use the `postulate` keyword in Agda.

```
postulate
  exch : Currency → Currency → PR ℤ
  disc : Currency → PR Bool × PR ℤ → PR ℤ
  snell : Currency → PR Bool × PR ℤ → PR ℤ
  absorb : Currency → PR Bool × PR ℤ → PR ℤ
```

Next, we implement (A1) - (A5). Recall from chapter 2 that these are axiom-like relations that arise from the nature of financial systems so we can only postulate them and not define them.

```
postulate
  A1 : ∀ (k : Currency) → exch k k ≡ K (+ 1)
  A2 : ∀ (k₁ k₂ k₃ : Currency) → liftPR₂ _*_ (exch k₂ k₁) (exch k₃ k₂) ≡ (exch k₃ k₁)
  A3 : ∀ (k : Currency) → ∀ (p : PR ℤ) → disc k (K true , p) ≡ p
  A4 : ∀ (k₁ k₂ : Currency) → ∀ (o : PR Bool) → ∀ (p : PR ℤ) →
    liftPR₂ _*_ (exch k₁ k₂) (disc k₂ (o , p)) ≡ disc k₁ (o , ( liftPR₂ _*_ (exch k₁ k₂)
    p))
  A5 : ∀ (k : Currency) → ∀ (o : PR Bool) → ∀ (p₁ p₂ : PR ℤ) →
    disc k (o , (liftPR₂ _+_ p₁ p₂)) ≡ liftPR₂ _+_ (disc k (o , p₁)) (disc k (o , p₂))
```

We can now implement the valuation semantics.

3.6 Implementing the Valuation Semantics

The valuation semantics are implemented as functions that take in a contract or observable and return a value process. We shall refer to the valuation semantics function for contracts as `eval` and that for observable as `val`.

```
eval : Currency → Contract → PR ℤ
eval k zeroC = K (+ 0)
eval k (one k₂) = exch k k₂
eval k (give c) = liftPR -_ (eval k c)
eval k (and c d) = liftPR₂ _+_ (eval k c) (eval k d)
eval k (or c d) = liftPR₂ _⊔_ (eval k c) (eval k d)
eval k (scale o c) = liftPR₂ _*_ (val o) (eval k c)
eval k (cond o c₁ c₂) = liftPR₃ if_then_else_ (val o) (eval k c₁) (eval k c₂)
eval k (when o c) = disc k ((val o), (eval k c))
eval k (anytime o c) = snell k ((val o), (eval k c))
eval k (until o c) = absorb k ((val o), (eval k c))


val : {a : Set} → Obs a → PR a
val (konst x) = K x
val (lift f o) = liftPR f (val o)
val (lift₂ f o₁ o₂) = liftPR₂ f (val o₁) (val o₂)
val date = datePR
```

Note that the `_⊔_` operator refers to the maximum value between two integers.

Chapter 4

Critical Evaluation

4.1 Proofs and Reasoning

In this chapter, we present examples of proofs using the valuation semantics implemented in chapter 3. We write these proofs in Agda and show how our implementation allows us to formally verify these proofs.

4.1.1 Equality

The valuation semantics defined translate contracts to value processes. However, to compare value processes we must define what it means for two value processes to be equal.

We define two value process to be equal when for all times, they return the same value. We may write this as follows.

For any type `a : Set`, two value process `p q : PR a` are said to be equal if and only if for all dates `d : Date`, `p d = q d`.

We can also encode this as the following type in Agda.

```
∀ (a : Set) → ∀ (p q : PR a) → ∀ (d : Date) → p d ≡ q d
```

For the proofs below, we do not make explicit use of this type but rather encode it within a larger type that includes the claim we are trying to prove. With this definition in mind, we are now able to write proofs as shown in the upcoming sections.

4.1.2 Commutativity

Recall the definitions of the `and` and `or` combinators for contracts.

```
and : Contract → Contract → Contract
or : Contract → Contract → Contract
```

Their valuation semantics were given as follows.

```
eval k (and c d) = liftPR₂ _+_ (eval k c) (eval k d)
eval k (or c d) = liftPR₂ _⊔_ (eval k c) (eval k d)
```

We understand that the addition operation and the maximum function are commutative in nature.

```
a + b = b + a
a ⊔ b = b ⊔ a or max(a, b) = max(b, a)
```

Therefore, intuition states that the `and` and `or` combinators must be commutative as well. Specifically, we assert the following.

```
and c1 c2 = and c2 c1
or c1 c2 = or c2 c1
```

Let us first treat the proof for commutativity of `and`. We declare the term `and-comm` as follows.

```
and-comm : ∀ (k : Currency) → ∀ (c d : Contract) → ∀ (t : Date) →
  (eval k (and c d)) t ≡ (eval k (and d c)) t
```

Therefore, the type of `and-comm` claims that for any currency, any pair of contracts `c` and `d` and any date `t`, the valuation semantics for `and c d` and `and d c` yield value process which return the same value for that date `t`.

We shall now supply a proof for this term. We write the following.

```
and-comm k c d t =
  begin
    (eval k (and c d)) t
  ≡⟨⟩
    (liftPR₂ _+_ (eval k c) (eval k d)) t
  ≡⟨⟩
    (eval k c t) + (eval k d t)
  ≡⟨ (+-comm (eval k c t) (eval k d t)) ⟩
    (eval k d t) + (eval k c t)
  ≡⟨⟩
    (liftPR₂ _+_ (eval k d) (eval k c)) t
  ≡⟨⟩
    (eval k (and d c)) t
  ∎
```

We start with `(eval k (and c d)) t` and apply the definition of `eval` and `liftPR`$_2$ to get `(eval k c t) + (eval k d t)`. Here we use the term `(+-comm (eval k c t) (eval k d t))` which arises in the `Data.Integer.Properties` library imported earlier. This term is proof that addition is commutative for integers and we apply the terms `(eval k c t)` and `(eval k d t)` to obtain `(eval k d t) + (eval k c t)`. After that, it is simply a matter of applying the definitions of `liftPR`$_2$ and `eval` in reverse to obtain the desired result.

We find that Agda's type checker accepts our proof. Therefore, the commutativity of `and` follows.

Note that the proof consisted of applying suitable definitions, an underlying algebraic property such as commutativity of addition over integers and then applying the same definitions in reverse. We shall see this as a recurring theme.

The proof for the commutativity of `or` is quite similar.

```
or-comm : ∀ (k : Currency) → ∀ (c d : Contract) → ∀ (t : Date) →
  (eval k (or c d)) t ≡ (eval k (or d c)) t
or-comm k c d t =
  begin
    (eval k (or c d)) t
  ≡⟨⟩
    (liftPR₂ _⊔_ (eval k c) (eval k d)) t
  ≡⟨⟩
    (eval k c t) ⊔ (eval k d t)
  ≡⟨ (⊔-comm (eval k c t) (eval k d t)) ⟩
    (eval k d t) ⊔ (eval k c t)
  ≡⟨⟩
    (liftPR₂ _⊔_ (eval k d) (eval k c)) t
  ≡⟨⟩
    (eval k (or d c)) t
  ∎
```

Here, we use `⊔-comm` which is a term that provides evidence that ⊔ or the maximum value function for two integers is commutative and we apply `(eval k c t)` and `(eval k d t)` to get the desired result.

4.1.3 Double Negation

The give combinator takes a contract and its rights and obligations and switches them to the other party. It makes intuitive sense that applying this combinator twice consecutively should yield the original contract. Specifically, the value processes for `give (give c)` should be equal to `c`.

Define `give-twice` as follows.

```
give-twice : ∀ (k : Currency) → ∀ (c : Contract) → ∀ (t : Date) →
  (eval k (give (give c))) t ≡ (eval k c) t
```

It is clear that the type of this term corresponds to the statement we are trying to prove. We present the following as proof.

```
give-twice k c t =
  begin
    (eval k (give (give c))) t
  ≡⟨⟩
    (liftPR -_ (eval k (give c))) t
  ≡⟨⟩
    - (eval k (give c) t)
  ≡⟨⟩
    - ((liftPR -_ (eval k c)) t)
  ≡⟨⟩
    - ( - ((eval k c) t))
  ≡⟨ (neg-involutive ((eval k c) t)) ⟩
    (eval k c) t
  ■
```

The key observation to make here is the use of the term `neg-involutive ((eval k c) t)` which is evidence that a double negative on an integer is a positive on the same integer. The claim follows from the result we obtain.

4.1.4 Distributivity

Consider the following claim.

The contract `and c1 (or c2 c3)` is equivalent to the contract `or (and c1 c2) (and c1 c3))`, given arbitrary contracts `c1`, `c2` and `c3`.

One can interpret this as the `and` combinator distributing over the `or` combinator.

In the language of contracts, this claim represents the idea that combining a contract *c1* and an *option* between contracts *c2* and *c3* yields the same contract as the *option* where we choose between the *aggregates* of *c1* and *c2* or *c1* and *c3*. We trust that the reader finds this interpretation convincing. But if not, we shall now present a proof for the same.

Define the following term.

```
and-distrib-or : ∀ (k : Currency) → ∀ (c_1 c_2 c_3 : Contract) → ∀ (t : Date) →
  (eval k (and c_1 (or c_2 c_3))) t ≡ (eval k (or (and c_1 c_2) (and c_1 c_3))) t
```

The type signature takes the claim as stated above and applies valuation semantics as needed to allow us to define the equality. We present the proof as follows.

```
and-distrib-or k c_1 c_2 c_3 t =
  begin
    (eval k (and c_1 (or c_2 c_3))) t
  ≡⟨⟩
    ((eval k c_1) t) + (((eval k c_2) t) ⊔ ((eval k c_3) t))
  ≡⟨ mono-≤-distrib-⊔ ( +-mono^r-≤ ((eval k c_1) t)) ((eval k c_2) t) ((eval k c_3) t) ⟩
    (((eval k c_1) t) + ((eval k c_2) t)) ⊔ (((eval k c_1) t) + ((eval k c_3) t))
  ≡⟨⟩
    ((liftPR_2 _+_ (eval k c_1) (eval k c_2)) t) ⊔ ((liftPR_2 _+_ (eval k c_1) (eval k c_3))
    t)
  ≡⟨⟩
    ((eval k (and c_1 c_2)) t) ⊔ ((eval k (and c_1 c_3)) t)
  ≡⟨⟩
    (liftPR_2 _⊔_ (eval k (and c_1 c_2)) (eval k (and c_1 c_3))) t
  ≡⟨⟩
    (eval k (or (and c_1 c_2) (and c_1 c_3))) t
  ■
```

Before we discuss the proof, we briefly mention that the addition operation conserves inequalities. This means that a ¡= b if and only if $a + c$ ¡= $b + c$ for integers a, b and c. This is the underlying property that implies that we can add the same integer to two other integers and the maximum between them stays unaffected.

In the proof above, similar to the ones discussed earlier, we draw our attention to the usage of the term `mono-≤-distrib-⊔ ( +-mono`^{r}`-≤ ((eval k c`$_1$`) t)) ((eval k c`$_2$`) t) ((eval k c`$_3$`) t)`. We reach into the depths of the Agda standard library to find `+-mono`^{r}`-≤ ((eval k c`$_1$`) t))` which is a term that provides evidence that the addition operator conserves inequalities as mentioned in the previous paragraph, in this case using the operand `(eval k c`$_1$`) t`. Then the term `mono-≤-distrib-⊔` is what provides evidence for the distributivity and takes in the specific operands as shown above. This property as a whole is what underpins the distribution of addition over the maximum function and then it is again simply a matter of applying definitions of `eval` and `liftPR`$_2$. The desired result follows and so does our claim.

As a whole, note that the common theme between these proofs is the use of simple algebraic properties that we already are aware of. Therefore, this shows how such proofs can be written simply by using these algebraic properties in conjunction with the valuation semantics.

4.2 Evaluating Agda

Agda has an expressive and powerful type system. This type system allows us to define a variety of relations that we can then write a formal proof for as shown throughout this paper. However, there do exist some limitations. Notably, Agda's solution to avoiding Russell's Paradox was by using Universe Levels. When implementing an observable, it disallowed us from implementing it as a Set and required a Set$_1$ instead. However, for the most part, this was not a barrier but rather an intuitive defect.

Looking back on the implementation, we made use of the Agda standard library. Whilst the properties we used were simple, there is support for stronger and complicated theorems. However, it was difficult to find what you require from the library at times and it would been helpful to have better signposting built in.

An interesting feature of Agda is that it makes sure functions cannot crash and always terminate which is not a feature is most other functional programming languages. Whilst restrictive, it is a great feature as it helps to ensure totality of functions and reduce human error. This also ties into Agda's feature of being in close proximity to the original mathematical definitions, functions and equational reasoning. It certainly complemented my mathematical background and strengthened my intuition.

The largest downside of using Agda is the lack of sufficient online support. This means that solving problems with your program ends up becoming extremely difficult and time consuming as you very rarely find direct solutions on the internet. Especially when learning the language for the first time, it compounds the difficulty.

Chapter 5

Conclusion

We conclude by reiterating on the contributions of this paper.

- An accessible introduction to Agda including a discussion on dependent types, defining terms and combinators, defining functions and theorem proving
  - Managed to discuss these with extended examples
  - Special focus was given to ensure the introduction was self-contained and easy to understand and follow
  - Showcased how to prove theorems in Agda
  - Unfortunately, due to space constraints, could not explain the mechanisms of how proofs work but it is not necessary and the reader should be able to practically prove theorems themselves
- An exposition of the combinator library and valuation semantics defined in the composing contracts paper
  - Through extended examples, took the ideas and definitions from the original contracts paper and delivered a thorough and strongly intuitive exposition of the original ideas
- A comprehensive and concrete implementation of the combinators as well as their valuation semantics in Agda
  - Used dependent and parametric types in Agda to write a concrete and almost complete implementation
  - Made some simplifications, but have acknowledged and justified them in the beginning
- A demonstration and examples on how our Agda implementation allows the user to write proofs of equalities that the combinators satisfy
  - Presented simple and intuitive examples so that the reader not only understands how to write the proofs but also knows where they arise from
  - Equips the reader to be able to come up with and write proofs of their own properties using the library implemented in Agda
- Support for the claim that the simple semantics by design give rise to simple algebraic properties
  - Using concrete examples of thorough proofs, it becomes clear how the design of the semantics gives rise to proofs using simple algebraic properties

Bibliography

[1] Agda Developers. Agda. URL: https://agda.readthedocs.io/.

[2] William L. Hosch. Peano axioms. *Encyclopedia Britannica*, 2010.

[3] Yannis Kassios. Formal proof. Technical report, University of Toronto, 2009.

[4] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1980.

[5] Ulf Norell. *Dependently Typed Programming in Agda*, pages 230–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-04652-0_5.

[6] Simon Peyton Jones and Eber Jean-Marc. *How to Write a Financial Contract*, pages 105–129. 01 2003. doi:10.1007/978-1-349-91518-7_6.

[7] Simon Peyton Jones, Eber Jean-Marc, and Julian Seward. Composing contracts: an adventure in financial engineering - functional pearl. 08 2000. doi:10.1145/351240.351267.

[8] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. August 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.