Decomposing Contracts

A Formalism for Arbitrage Argumentations

Steffen Schuldenzucker

Born 1988-09-23 in Bonn, Germany

2014-09-05

Master's Thesis Mathematics

Advisor: Prof. Dr. Stefan Geschke

Hausdorff Center for Mathematics

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Contents

1 Introduction

Arbitrage arguments are statements about the prices of derivatives in *perfect financial markets* which are based purely on the assumption that no market participant can make profit without exposing herself to risk.[1] Hence, arbitrage arguments can be made without any stochastic assumptions. Examples for arbitrage arguments are forward prices of dividend-free shares, the put-call-parity and Merton's theorem about American options.[2]

A "perfect" financial market is here defined by the following properties:

- All market participants have equal access to the market.
- All market participants have equal access to information.
- No market participant has transaction costs.
- All assets are perfectly liquid, i.e. can be acquired in arbitrary amounts at any time.
- Prices are driven purely by the principles of supply and demand.
- No time is required for communication.

An important consequence, and an assumption in the following, is that there is no *BID-ASK spread*: Any contract[3] which is traded can be bought and sold for the same price, making the concept of "the" price reasonable in the first place. A special case are interest rates: If an interest rate is viewed as the price of future money, it follows that the rate for borrowing and the rate for investing money are equal and the same for everyone.

A traditionally central concept is the "present value" of a contract. The idea is that there exists a certain "fair price" which incorporates any possible future payments with their probabilities. It is the price a risk-neutral trader "should" be willing to pay. It is then assumed that this price is in fact attained by the market. Computing present values is a difficult task and usually done using heavy distributional assumptions in a stochastic model. For example, the binomial model assumes that the price of a share can only increase or decrease by a certain factor in each time step while the Black-Scholes-Merton model assumes that a share price behaves basically like a Brownian motion.[4]

What is common to all these models is that they should not allow arbitrage in order to be elementary well-formed: If one can show that a contract y is "preferable" to a contract x "in arbitrage", i.e. that one can arrive at a risk-free position from buying x and selling y, then the assigned present value of x should be less or equal to that of y.

[1] Such an operation is called *arbitrage* and consequently the lack of arbitrage is called *no-arbitrage condition*. I use the term "arbitrage" only for *deterministic* arbitrage, as opposed to *statistical* arbitrage where a profit might be only made in expectation.

[2] Knowledge of finance is not required to understand this thesis. All concepts will be introduced as required, however for some, I only give formal definitions. All arbitrage statements in this thesis are taken from [1], the first pages of which can also serve as an introduction to financial derivatives.

[3] I use the terms "financial derivative", "asset" and "contract" interchangeably, so a contract can really be anything finance is concerned with.

[4] The two models can be found in [1] and [2].

My approach is to define *not* another way to compute present values, but rather the "preferable in arbitrage" relation, axiomatically: While a present value states that the price of a financial asset should be equal to that present value, the *present value relation* ("preferable in arbitrage", a partial ordering of contracts) only states that prices should reflect the relation. So the latter notion is weaker and hence allows more statements to be made and/or use weaker assumptions. At the same time, it is a true generalization: Whenever it can be shown by means of arbitrage arguments that a contract has a present value, this can be expressed by saying that the contract is both preferable to the contract that just pays the present value (a certain amount of money) and vice versa. However, contracts for which no present value is known in general or for which it might be even known that there *cannot* be a present value can still be compared. Relatedly, I will show inside the framework that if two contracts that have a price[5] are related by a present value relation (one is preferable to the other), then prices must reflect this (one must be greater or equal to the other).

My thesis establishes the following:

1. A framework in many-sorted (or "typed") first-order logic (*MSL*) to define financial contracts as well as market data and market conditions formally, by introducing fundamental "building blocks" or "combinators" (sections 2 and 3).

   The approach is then always holistic in that not only certain classes of derivatives such as options or swaps are supported, but a general mechanism is provided describing the behavior of the building blocks.

2. A relation "$\preceq_.$" where $x \preceq_b y$ should mean that "y is preferable to x in arbitrage under conditions b" and axioms which relate the different combinators (section 3.1). These axioms will reflect the fundamental arguments in arbitrage reasoning. I call the theory resulting from this and the previous point *LPT*, the Logic Portfolio Theory of arbitrage-free markets.

   LPT is split into three layers: Primitive data types and operations (LPT_{Prim}, discussed in this section and appendix A.3), *observables* as a means to express market data (LPT_{Obs}, section 2) and the theory of contracts itself (section 3).

3. Evidence that the framework does indeed capture the informal notion of arbitrage arguments by proving some well-known statements inside the theory (section 4).

4. The proof that stochastic market models are indeed models of the theory as long as certain restrictions are made (section 5). – A generalized version of the binomial model[2, p. 249] is supported while a Wiener process in continuous time[2, p. 271] is not yet.

My work is based on two papers by Simon Peyton Jones and Jean-Marc Eber [3, 4] in which they develop the formal system of observables and contracts as a programming library in the Haskell [5] language. Such a library is essentially a

[5] I use the terms "present value" and "price" interchangeably here. That is because a present value of a traded asset that can be computed by arbitrage arguments only must be equal to the price. – Otherwise, there is an arbitrage opportunity.

formal language and I was able to re-use the approach from [3] with some small modifications.[6]

The aim of the two papers is computing present values while I aim for the relations between contracts. Peyton Jones' and Eber's work does not provide any axioms describing the behavior of the primitives introduced. Peyton Jones and Eber *do* mention that one can derive rules from their provided stochastic interpretation, but as such rules must be based on a certain class of market models, they should not be called pure "arbitrage arguments".

Haskell is a functional language and hence, following Haskell's style, my formalism heavily relies on functions. I introduce some syntactic modifications to MSL which I call *lambda notation* to denote functional constructions easily while staying first-order. I give a short overview of my MSL variant in section 1.2 below and the full definition can be found in appendix A.1.1.

As a by-product, I provide a way to translate a Haskell program into MSL (section A.2) as long as certain restrictions are made as well as a way to model Haskell's algebraic data types (ADTs) in a probabilistic setting (appendix D).

This thesis should be viewed primarily as an exercise in design: I show how a collection of common sense concepts and arguments can be be condensed into a solid and *abstract* mathematical framework without imposing a particular mathematical interpretation on what – in this case – a financial contract "really" is.

Remark 1.1 (A note on style). In the parts of the sections 2 and 3, where the LPT theory is constructed, I first introduce new primitives and/or axioms, then prove some lemmas about them. That is, axioms are introduced together with their motivation and consequences instead of all in one place.

Axioms are marked by an asterisk, e.g.

$$\texttt{return} \circ f = \texttt{fmap}\ f \circ \texttt{return} \qquad (*\text{Mo1})$$

$$\top \neq \bot \qquad (*2.1)$$

I will now continue by giving an example of an arbitrage argument, then an introduction to the formal framework used in this thesis.

1.1 Example arbitrage argument: Put-call parity

To give an impression of what, and why, we want to formalize, consider the put-call-parity [1, sec. 10.4] as a non-trivial example of an arbitrage argument.

We first need to define what a "put" and a "call" are:

Definition 1.2 (European Options, informal)**.** Let S be a dividend-free share[7] and let $K \in \mathbb{R}^+$. Let T be a point in time. Assume that all amounts are paid in a certain currency, say USD.

A *European call option* is the derivative contract that grants the holder the right, but not the obligation, to buy S for price K at time T (which is assumed

[6] Knowledge of Haskell is not required for being able to read this thesis, except for the Haskell-centered sections A.2 and D, of course. However, those familiar with Haskell will recognize well-known design patterns, most prominently that of a *monad*. A very brief overview of the core ideas of the language can be found at the beginning of section A.2.

[7] which is – of course – a company share which is known to not pay a dividend in the relevant time period.

to lie in the future). A *European put option* is the contract that grants the holder the right to sell S for K instead.

It is clear that European options must have non-negative value because it is not possible to make a loss from them. It is further easy to see that the payout of a European call option at time T is

$$[P_T(S) - K]^+$$

where $P_T(S)$ is the share price of S at time T and where $[x]^+ = \max(0, x)$. Likewise, the payout of the respective put option at time T is

$$[K - P_T(S)]^+.$$

Theorem 1.3 (Put-call parity, informal)**.** *Let S, K and T be as above and fix a point in time $t \leq T$. Let r be the risk-free interest rate.*[8] *Let C and P be the prices of the European call- and put option, respectively, and let $P_t(S)$ be the price of S at time t. Then the following equality holds at time t:*

$$C + (1+r)^{-(T-t)} \cdot K = P + P_t(S)$$

Proof. I give two proofs here. Both are essentially taken from [1, sec. 10.4].

For the first, note that the LHS is the cost of receiving at time T

$$[P_T(S) - K]^+ + K = \max(P_T(S), K)$$

by the payout of the call as discussed above and the fact that $(1+r)^{-(T-t)} \cdot K$ invested at rate r over a time of $T-t$ yields $(1+r)^{T-t} \cdot (1+r)^{-(T-t)} \cdot K = K$. Likewise, the RHS is the cost of receiving at time T

$$[K - P_T(S)]^+ + P_T(S) = \max(K, P_T(S)).$$

As the two payouts are equal, the prices must be equal as well.

As a second proof, I give an explicit construction of an arbitrage portfolio for the "$<$" case. The "$>$" case is symmetric. So assume that we have "$<$" in the above formula, i.e.

$$C + (1+r)^{-(T-t)} \cdot K - P - P_t(S) < 0.$$

Then the portfolio from figure 1 leads to immediate profit ("free lunch"). As the balance at time T is 0, it does not expose the trader to risk. Hence, this is arbitrage. □

There are now a few questions that should be asked about the preceding proof:

1. What is a "dividend-free share", after all? What are the properties which are used implicitly in this proof?

   In the arbitrage portfolio, it is assumed that the trader executing the strategy owns S or, alternatively, that shares can be "borrowed", so it is

[8] I.e. the rate at which money can both be invested and borrowed while being sure to get the money back. Existence of such a rate is an assumption (cf. above).

Figure 1 Arbitrage portfolio for theorem 1.3

Action at t	Balance at t	Balance at T
Buy the call	$-C$	$[P_T(S) - K]^+$
Invest $(1+r)^{-(T-t)} \cdot K$	$-(1+r)^{-(T-t)} \cdot K$	K
Sell the put	P	$[K - P_T(S)]^+$
Sell the share, buy back at T	$P_t(S) - S$	$S - P_T(S)$
Total	> 0 dollars $-S$	$+S$

always possible to short sell. The trader would then get back the share, and there is no dividend she could miss.

However, it needs to be known that it is in fact *desirable* to get S back. For example, if it is known at time t that S will reach a strong peak between t and T, then turn worthless at time T, figure 1 is not an arbitrage portfolio. Of course, in this case the price of S would already adjust at time t to reflect the future price change. Relatedly, the portfolio only lists the balances at time t and T, but does not mention the opportunity to sell in between which the trader lets go in order to execute the strategy.

The point is that S cannot be replaced by any other contract here. At the same time, it is clear that the put-call parity works for other underlyings as well, such as foreign currencies if one equates for the foreign interest rate. One can also essentially replace S by an interest rate to receive the *cap-floor parity* [1, sec. 28.2].

2. Is the fixed risk-free interest rate r actually required?

   This is actually three related points: First, there is not in general a single "natural" risk-free rate that should be used for r [1, sec. 4.1]. Second, r does not depend on $T - t$ here, which is not realistic.[9] Third, r cannot change over time here, which is not realistic either.

3. What is then the core of the argument after all? Are there any other assumptions made implicitly?

I will show the put-call parity formally in section 4.5. This will lead to a characterization of dividend-free shares (section 4.7) and we will see that all questions from the second point can be answered "not required". I will discuss interest rates formally in section 4.2. For the third point, the axiomatic approach guarantees that all assumptions are mentioned explicitly.

Note that none these are novel! Each of the above three points can be resolved by careful inspection of the above proof. My approach however makes it *easy* keep the statement and the proof as general as possible.

1.2 Introduction to the formal framework used

The following section is a summary of appendix A, which should be consulted for details.

As mentioned above, I use many-sorted first-order logic (*MSL*) as the formal framework in which all argumentation happens. MSL is essentially the same as

[9] Cf. "Term structure of interest rates" in [1].

regular first-order logic where every object or symbol has an associated *type* and types must match when symbols are combined. In order to support a functional style, I define a custom way to denote functions which I call *lambda notation*. Formulas, proofs and models then find an exact analog to first-order logic.

Notation 1.4.

1. A *type* is either a primitive type (or *sort*) like $\mathbb{Z}$ or `Obs Bool` (constants and variables), a functional type like $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ (functional symbols, lambda terms), or a relational type like $\mathscr{R}(\mathbb{Z}, \mathbb{Z})$ (relational symbols).

2. In the previous point, `Obs` is a type constructor, i.e. for any sort a, there is a sort `Obs` a. This is just an ordinary sort name, defined by string concatenation, without any special meaning. However, a certain set of functions will be defined on all sorts of form `Obs` a.

3. $\mathbb{Z}$, $\mathbb{R}$ etc. are just names of sorts here. Their subset relations are modeled explicitly (cf. section A.3.2), but are treated intuitively.

4. I write $t :: \alpha$ to state that t has type α. The framework assumes that everything has a type attached, but in practice, I usually leave the types out.

5. Application is denoted by juxtaposition, i.e. I write $f\ x\ y$ instead of $f(x, y)$.

6. Application is done one argument after the other: $f\ x\ y$ in fact means $(f\ x)\ y$: Applying less arguments than the function takes yields a new function in one argument less.

7. I write $\lambda\, x :: s.\ t$ for the function in one argument x of type s that is defined by the term t (which may contain x). Functions in several arguments can be defined by $\lambda\, x_1.\ \lambda\, x_2.\ \ldots \lambda\, x_n.\ t$ or short by $\lambda\, x_1\ \ldots\ x_n.\ t$.

   When defining functions I also write

   $$f\ x\ y := \ldots$$

   instead of $f := \lambda\, x\ y.\ \ldots$.

8. Application of a lambda term to a term is done by replacing the parameter by the argument, i.e. $(\lambda\, x.\ t)\ t' := t[x/t']$.

9. A function an argument of which is itself of functional type is called *higher-order*. This is not actually allowed, but one can emulate the behavior of higher-order functions by a technique I call *closure emulation* (section A.1.3).

   Higher-order functions are different from regular first-order functions because MSL is a first-order logic: Functions are not objects, so functions cannot actually appear as parameters. One is merely able to talk about lambda terms, which is what the closure emulation schema does in a systematic way.

Example 1.5. Now the following is meaningful:[10]

Given sorts $\mathbb{Z}$ and $\mathbb{R}$ and symbols $(-)_{\mathbb{Z}} :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$, $(-)_{\mathbb{R}} :: \mathbb{R} \to \mathbb{R} \to \mathbb{R}$, $\texttt{floor} :: \mathbb{R} \to \mathbb{Z}$ and $\texttt{asReal} :: \mathbb{Z} \to \mathbb{R}$, the following are functional terms (partly using my short notation):

- $t_1 := \lambda\, x\ y.\ (-)_{\mathbb{Z}}\ (x :: \mathbb{Z})\ (y :: \mathbb{Z})$
- $t_2 := \lambda\, (y :: \mathbb{R})\ (x :: \mathbb{R}).\ (-)_{\mathbb{R}}\ x\ y$
- $t_3 := \lambda\, x.\ (-)_{\mathbb{R}}\ (\texttt{asReal}\ (\texttt{floor}\ x))\ x$
- $t_4 := \lambda\, x.\ (+)_{\mathbb{R}}\ (t_3\ x)\ x$

The types are $t_1 :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$, $t_2 :: \mathbb{R} \to \mathbb{R} \to \mathbb{R}$, $t_3 :: \mathbb{R} \to \mathbb{R}$ and $t_4 :: \mathbb{R} \to \mathbb{R}$.

From the names of the functions, one would expect that `floor` and `asReal` come with axioms such that $t_4\ x = x$ for any x. – Or short $t_4 = \texttt{id}$.

Notation 1.6 (Polymorphism)**.** One often wants to define the same function for many different types. For example, the function

$$\texttt{square} := \lambda\, x.\ x \cdot x$$

makes sense for $x :: \mathbb{Z}$, $x :: \mathbb{R}$ etc. This can be solved by implicitly thinking "$(\cdot)$" to stand for many different function symbols $(\cdot)_{\mathbb{Z}} :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$, $(\cdot)_{\mathbb{R}} :: \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ etc. and receiving many functions $\texttt{square}_{\mathbb{Z}}$, $\texttt{square}_{\mathbb{R}}$ etc. – which are all called just `square` of course. In particular, the above functions $(-)_{\mathbb{Z}}$ etc. would just be written $(-)$.

If the types are arbitrary, I use lower-case type variables as in

$$\texttt{return} :: a \to \texttt{Obs}\ a$$

from section 2: For any a, there is (a sort $\texttt{Obs}\ a$ and) a function $\texttt{return}_a$, but they are all called just `return`.

Example 1.7 (Higher-order function application). Consider the higher-order function

$$\texttt{fmap} :: (a \to b) \to \texttt{Obs}\ a \to \texttt{Obs}\ b$$

from section 2. For the moment, it is only important that `fmap` accepts a function in one argument for any combination of argument- and result types.

Then, whenever a is a numeric type, the following is well-defined:

$$\lambda\, (i :: a)\ (io :: \texttt{Obs}\ a).\ \texttt{fmap}\ (\lambda\, j.\ i + j)\ io$$

Note how here, $(+) :: a \to a \to a$ and hence $(\lambda\, j.\ i + j) :: a \to a$. So the expression fits the type of `fmap` with $b = a$.

Remark 1.8 (Closures). The defining term of a lambda expression which is passed as an argument to a higher-order function may contain variables which are not arguments of that lambda expression, but only in scope outside. The lambda is then called a *closure* storing the variables in question as *closure context*. For example, in the above example, the variable i would be closure context: It occurs in $(\lambda\, j.\ i + j)$, but is only in scope outside: i is used to construct the function passed to `fmap`.

[10] This example is the same as A.10.

As indicated by the name, care is taken to have the closure emulation schema support closures. Closures are also a core feature of functional programming languages like Haskell.

In the following, I assume that the basic data types and functions such as $\mathbb{R}$, $\mathbb{Z}$, $\mathbb{N}$, `Bool`, $(+)$ etc. as well as `Time` and `TimeDiff` types are given. These common data types and functions constitute the first part $\mathrm{LPT}_{\mathrm{Prim}}$ of my theory. Details can be found in appendix A.3.

2 Observables – Formalizing market data

This section defines the theory $\mathrm{LPT}_{\mathrm{Obs}}$, the *theory of observables*.

Objects of type `Obs` a, i.e. *observables* are "sources of market data" or "things that can change over time" of type a in the broadest sense. Conceptually, observables must be visible to everyone in the market and at each point in time, all market participants must agree on the value of an observable. – Hence the name. For example, insider information would *not* be observable.

Observables will be the only way the framework is going to be able to talk about things that can change, depend on conditions of the market etc. The other main component of the framework, contracts, do not change over time (but can access observables to read these pieces of data from the market).

The language of observables is crafted such they support any computation on market data as well as accessing its history, but not looking into the future.

- Most obviously, observables will model the prices of assets. These observables will have type `Obs` $\mathbb{R}$ or `Obs` $\mathbb{R}^+$.
- Any other piece of information, except for constants, that may occur in a contract must be given by an observable. For example, if one would want to formally analyze an insurance against drought in Peru, there should be a metric for that as an observable.
- A condition that may be true or false at any given point in time is of type `Obs Bool`.
- A special observable, `now` :: `Obs Time`, contains the current time.
- One can define a contract that "reads" an observable at acquisition and uses the resulting value by the `read′` and "$\rightsquigarrow$" operations defined section 3 below, thus making contracts "dynamic". Whenever payments depend on market data, e.g. variable interest yields, or sales of shares for their current price, these must be given by an observable.

The concept of an observable is taken from [3], the underlying concept of a *monad* came originally from category theory and has become a well known design pattern in functional programming.[11] In comparison to [3], I added the `ever` and `always` primitives from section 2.3 as well as all axioms, of course.

An observable of type `Obs` a is to be seen as an abstract *description* of a time-varying value. A concrete representation can be, for example:

- The set of functions `Time` $\to a$.
- The stochastic processes on a with respect to a certain filtration and a certain sample space. This should be the canonical model to keep in mind when discussing observables. One receives the trajectory model as a special case where the sample space is trivial, i.e. everything is deterministic.

  Peyton Jones and Eber use this interpretation in their paper.

[11] For example, Haskell's I/O system [5, sec. 7] is implemented as a monad which is consequently called `IO`. A classic paper on monads in functional programming is [6]. For a category-theoretic viewpoint, cf. remark 2.1. I define below what a monad is.

Figure 2 Primitives for observables

`return` :: $a \to$ `Obs` a
: `return` x has the value x at any point in time. Peyton Jones and Eber[3] write `konst` for `return`, but `return` is the standard name.

`fmap` :: $(a \to b) \to$ `Obs` $a \to$ `Obs` b
: `fmap` f o is the observable o with the function f applied to each value, at any point in time.

`join` :: `Obs` (`Obs` a) $\to$ `Obs` a
: `join` o reads, at any point in time, the observable o to receive a new observable of type `Obs` a, then reads that as well to receive a value of type a. `join` is listed here for its theoretical elegance and brevity. Most expressions use its equivalent cousin "$\gg\!=$" (*bind*, defined below) instead.

`now` :: `Obs Time`
: The current time.

- Sometimes it helps to imagine an observable as a group of small interconnected computers (or components of a piece of software) that receive any relevant market data over a network line. A component can store data in a limited way and perform calculations on it as well as pass its result on to superordinate components.
- Finally, the underlying concept of a monad can be thought as a form of computation. In the case of observables, a "command" would then mean accessing a certain piece of market data or its history or perform a calculation on the fetched values.

Fig. 2 lists primitive operations on observables I assume to exist with their desired meaning. This works in many-sorted first-order logic by adding for any sort a a new sort `Obs` a and function symbols $\texttt{return}_a$ and $\texttt{join}_a$ of the corresponding types. For `fmap`, being higher order, one needs to add as many symbols as it it permits first arguments, i.e. one needs to execute the corresponding *closure emulation schema* (cf. section A.1.3). For `now`, one only needs to add a single constant symbol. Section A.1.2 provides a systematic approach to such *modification operations*.

In the following, I will introduce axioms by which the primitives should be connected.

`fmap` should "just" apply a function inside an observable. Hence, one expects the following *functor laws* to hold for any sorts a, b, c and $g :: a \to b$ and $f :: b \to c$:

$$\texttt{fmap id} = \texttt{id} \qquad :: \texttt{Obs}\ a \to \texttt{Obs}\ a \qquad \text{(*Fu1)}$$

$$\texttt{fmap}\ (f \circ g) = \texttt{fmap}\ f\ \circ \texttt{fmap}\ g \quad :: \texttt{Obs}\ a \to \texttt{Obs}\ c \qquad \text{(*Fu2)}$$

Here, $\texttt{id} = \lambda\, x.\ x$. Define further

$$(\gg\!=) ::\ \ \texttt{Obs}\ a \to (a \to \texttt{Obs}\ b) \to \texttt{Obs}\ b$$
$$o \gg\!= f := \texttt{join}\ (\texttt{fmap}\ f\ o).$$

The intuitive meaning of "$\gg\!=$" is as follows: $o \gg\!= f$ reads, at any point in time, the observable o to receive a value $x :: a$. The function f is applied to x to receive a new observable $f\ x$ and that observable is read again to receive the result. Examples for how "$\gg\!=$" is used can be found in the following sections.

The following rules (which are *not* axioms here) are easily justified from the intuition of observables: For $o :: \texttt{Obs}\ a$, $x :: a$, $f :: a \to \texttt{Obs}\ b$ and $g :: b \to \texttt{Obs}\ c$ one expects the following:

$$o \gg\!= \texttt{return} = o \qquad :: \texttt{Obs}\ a \qquad \text{(Mo1')}$$

$$\texttt{return}\ x \gg\!= f = f\ x \qquad :: \texttt{Obs}\ b \qquad \text{(Mo2')}$$

$$(o \gg\!= f) \gg\!= g = o \gg\!= (\lambda x.\ f\ x \gg\!= g) \qquad :: \texttt{Obs}\ c \qquad \text{(Mo3')}$$

`join` can be expressed in terms of "$\gg\!=$": If $p :: \texttt{Obs}\ (\texttt{Obs}\ a)$, consider $p \gg\!= \texttt{id}_{\texttt{Obs}\ a}$. We have $p \gg\!= \texttt{id} = \texttt{join}\ (\texttt{fmap}\ \texttt{id}\ p) = \texttt{join}\ (\texttt{id}\ p) = \texttt{join}\ p$ via (*Fu1). It is not hard to show that the laws (Mo1')–(Mo3') are equivalent to the following *monad laws* which I chose as axioms due to their theoretical simplicity. For any sorts a and b and $f :: a \to b$, the following should hold:

$$\texttt{return} \circ f = \texttt{fmap}\ f \circ \texttt{return} \qquad :: a \to \texttt{Obs}\ b \qquad \text{(*Mo1)}$$

$$\texttt{join} \circ \texttt{fmap}\ (\texttt{fmap}\ f\ ) = \texttt{fmap}\ f \circ \texttt{join} \qquad \text{(*Mo2)}$$

$$:: (\texttt{Obs}\ (\texttt{Obs}\ a)) \to \texttt{Obs}\ b$$

$$\texttt{join} \circ \texttt{fmap}\ \texttt{join} = \texttt{join} \circ \texttt{join} \qquad \text{(*Mo3)}$$

$$:: \texttt{Obs}\ (\texttt{Obs}\ (\texttt{Obs}\ a)) \to \texttt{Obs}\ a$$

$$\texttt{join} \circ \texttt{return} = \texttt{id} \qquad :: \texttt{Obs}\ a \to \texttt{Obs}\ a \qquad \text{(*Mo4)}$$

$$\texttt{join} \circ \texttt{fmap return} = \texttt{id} \qquad :: \texttt{Obs}\ a \to \texttt{Obs}\ a \qquad \text{(*Mo5)}$$

Remark 2.1 (Connection to category theory). As the names suggest, axioms (*Fu1) and (*Fu2) state that $(\texttt{Obs}, \texttt{fmap})$ should form a *functor* and (*Mo1)–(*Mo5) state that $(\texttt{Obs}, \texttt{fmap}, \texttt{return}, \texttt{join})$ should form a *monad*.[12]

To be precise, if $\mathscr{A}$ is a model of the here-described theory $\mathrm{LPT}_{\mathrm{Obs}}$, then one can consider the category $\mathcal{C}^1_{\mathscr{A}}$ formed by the interpretations of sorts (as objects) and functional terms in a single parameter[13] (as morphisms) and where composition is given by chaining of lambda terms. This is a subcategory of $\mathcal{C}_{\mathscr{A}}$ from remark A.14.

Now consider the assignment $\texttt{Obs}^{\mathscr{A}}$ that maps any object $a^{\mathscr{A}}$ to $(\texttt{Obs}\ a)^{\mathscr{A}}$ and any morphism $(f :: a \to b)^{\mathscr{A}}$ to $(\texttt{fmap}\ f)^{\mathscr{A}}$. The axioms (*Fu1) and (*Fu2) state that $\texttt{Obs}^{\mathscr{A}}$ should be a functor from $\mathcal{C}^1_{\mathscr{A}}$ to itself. Traditionally, one would write here $\texttt{Obs}^{\mathscr{A}}\ f$ instead of $\texttt{fmap}\ f$.

For the second set of axioms, note how $\texttt{return}_a{}^{\mathscr{A}} : a^{\mathscr{A}} \to \texttt{Obs}^{\mathscr{A}}\ a^{\mathscr{A}}$ and $\texttt{join}_a{}^{\mathscr{A}} : \texttt{Obs}^{\mathscr{A}}\ (\texttt{Obs}^{\mathscr{A}}\ a^{\mathscr{A}}) \to \texttt{Obs}^{\mathscr{A}}\ a^{\mathscr{A}}$ are collections of morphisms in $\mathcal{C}^1_{\mathscr{A}}$, one per object. Axioms (*Mo1) and (*Mo2) state that these collections should form two natural transformations $\texttt{return} : I \to \texttt{Obs}^{\mathscr{A}}$, where I is the identity functor mapping any object and morphism to itself, and $\texttt{join} : (\texttt{Obs} \circ \texttt{Obs}) \to \texttt{Obs}$.

[12] For the category-theoretic concepts mentioned here, cf. [7]: Chapter I for functors and natural transformations and chapter VI for monads.
Their knowledge might prove helpful in the following, but is by no means required.

[13] Note that this is not really a restriction because there are tuples.

Figure 3 Haskell code in `do` notation and equivalent function definition

```
f :: Obs Int -> Obs Int
          -> Obs Int
f o p = do x <- o
           y <- p
           return (x + y)
```

$$
\begin{aligned}
f :: \ & \texttt{Obs}\ \mathbb{Z} \to \texttt{Obs}\ \mathbb{Z} \\
& \to \texttt{Obs}\ \mathbb{Z} \\
f\ o\ p := \ & o \gg\!= \lambda x. \\
& p \gg\!= \lambda y. \\
& \texttt{return}\ (x + y)
\end{aligned}
$$

Axioms (*Mo3), (*Mo4) and (*Mo5) now state that $\texttt{Obs}^{\mathscr{A}}$ should form a monoid where the domain set is replaced by the functor $\texttt{Obs}^{\mathscr{A}}$, the cartesian product is is replaced by composition of functors, `return` is the neutral element and `join` is multiplication: (*Mo3) is associativity and (*Mo4) and (*Mo5) are left and right identity, respectively. Traditionally, `return` would be called η and `join` would be called μ.

In section 5, I will define a $\mathrm{LPT}_{\mathrm{Obs}}$-model where $\mathcal{C}^1_{\mathscr{A}}$ is a subcategory of the category $\mathcal{M}$ of measurable spaces and functions and $\texttt{Obs}^{\mathscr{A}}$ is in fact a (restriction of a) monad on $\mathcal{M}$.

A point to keep in mind is that the formalism allows the argument of `fmap` to be *partially applied* as in $\lambda x\ o.\ \texttt{fmap}\ (+x)\ o$: Here, the function $(+)$ expects two arguments, but is only applied one to yield a new function in one argument which is then passed to `fmap`. The closure emulation schema makes sure that this notation is meaningful in MSL and I will show that the model from section 5 supports it.

A common pattern when working with monads are "chains" of "$\gg\!=$" calls. These are so common even that Haskell offers its own syntax for them, namely `do`-notation (figure 3). Note how `do`-notation visually expresses the idea that monads describe an "abstract computation" as addressed above.

2.1 Higher lifts

From "$\gg\!=$" and `return`, one can define generalize `fmap`, which *lifts* a function in a single parameter to observables, to any number n of arguments:

$$
\begin{aligned}
\texttt{lift}_n :: \ & (a_1 \to \ldots \to a_n \to b) \\
& \to \texttt{Obs}\ a_1 \to \ldots \to \texttt{Obs}\ a_n \to \texttt{Obs}\ b \\
\texttt{lift}_0\ x = \ & \texttt{return}\ x \\
\texttt{lift}_{n+1}\ f\ o_1\ \ldots\ o_{n+1} = \ & o_1 \gg\!= \lambda x_1.\ \texttt{lift}_n\ (f\ x_1)\ o_2\ \ldots\ o_{n+1} \\
= \ & o_1 \gg\!= \lambda x_1.\ o_2 \gg\!= \lambda x_2.\ \ldots o_{n+1} \gg\!= \lambda x_{n+1}. \\
& \texttt{return}\ (f\ x_1\ \ldots\ x_{n+1})
\end{aligned}
$$

where $f\ x_1 = \lambda x_2\ \ldots\ x_{n+1}.\ f\ x_1\ x_2\ \ldots\ x_{n+1}$. Note that $\texttt{lift}_1 = \texttt{fmap}$ and $\texttt{lift}_0 = \texttt{return}$.

Now the function from figure 3 could be written

$$\texttt{lift}_2\ (+).$$

The following lemmas are called "lift collapsing lemmas". They can all be shown by relatively simple, but lengthy calculations, which are given in appendix B.

One first notes that any sequence of "$\gg\!=$" where the bound variable is not used to construct the next observable is actually a lift. Note how *any* expression on observables looks like this at the moment or is equal to such an expression because there only are `return` and "$\gg\!=$". This will change in section 2.3.

Lemma 2.2. *Let* $f :: a_1 \to \ldots \to a_n \to \texttt{Obs}\ a$ *and* $o_i :: \texttt{Obs}\ a_i$ *for* $i = 1, \ldots, n$. *Then*

$$\begin{aligned} & o_1 \gg\!= \lambda x_1.\ o_2 \gg\!= \lambda x_2.\ \ldots o_n \gg\!= \lambda x_n.\ f\ x_1\ \ldots\ x_n \\ & \quad = \texttt{join}\ (\texttt{lift}_n\ f\ o_1\ \ldots\ o_n) \end{aligned}$$

Lemma 2.3 ((*Fu2) for higher arities / collapsing lifts)**.** *Let* $n \geq 0$, $m \geq 1$, $g :: a_1 \to \ldots \to a_n \to a$, $f :: a \to b_1 \to \ldots \to b_m \to b$ *and define*

$$\begin{aligned} f \circ_n g\ & :: a_1 \to \ldots \to a_n \to b_1 \to \ldots \to b_m \to b \\ f \circ_n g & := \lambda x_1\ \ldots\ x_n\ y_1\ \ldots\ y_m.\ f\ (g\ x_1\ \ldots\ x_n)\ y_1\ \ldots\ y_m \\ & \quad = \lambda x_1\ \ldots\ x_n.\ f\ (g\ x_1\ \ldots\ x_n). \end{aligned}$$

Then

$$\texttt{lift}_{m+1}\ f \circ_n \texttt{lift}_n\ g\ =\ \texttt{lift}_{m+n}\ (f \circ_n g).$$

The type of this function is

$$\texttt{Obs}\ a_1 \to \ldots \to \texttt{Obs}\ a_n \to \texttt{Obs}\ b_1 \to \ldots \to \texttt{Obs}\ b_m \to \texttt{Obs}\ b.$$

Lemma 2.4 ((*Mo1) for higher arities)**.** *Let* $n \geq 0$ *and* $f :: a_1 \to \ldots \to a_n \to a$. *Define*

$$\begin{aligned} & \texttt{lift}_n\ f \circ^n \texttt{return} ::\ \ a_1 \to \ldots \to a_n \to \texttt{Obs}\ a \\ & \texttt{lift}_n\ f \circ^n \texttt{return} := \lambda x_1\ \ldots\ x_n.\, \texttt{lift}_n\ f\ (\texttt{return}\ x_1)\ \ldots\ (\texttt{return}\ x_n)\,. \end{aligned}$$

Then

$$\texttt{return} \circ_n f\ =\ \texttt{lift}_n\ f \circ^n \texttt{return}.$$

Remark 2.5. One receives more general versions of lemma 2.3 and lemma 2.4 where the inner lift can occur at any position, not only as the first argument of the outer lift, by unrolling the recursive definition of $\texttt{lift}_n$.

By applying the lemma several times, it is clear that a chain of several levels of lifts, possibly at several positions, can always be collapsed into a single one and that a argument of form `return` x can always be removed.

For general monads, the ordering of lift arguments matters: For example, Haskell's `IO` monad encapsulates actions such as reading and writing files and sending data over a network. The result of such an action could be the number of bytes written or the trivial result () (*unit*, the unique element of the type (), also called unit, or 0-ary tuple). And clearly the ordering in which network packets are sent matters even if all one does is applying a lift to the result. So in general monads, lift arguments cannot be reordered or removed if results are not used in the lifted function.

Observables however should not "do" anything:[14] They just "read" market data and the ordering of reads does not matter, neither matters whether data is read once or twice or not at all if a piece of data is not used.[15] The following three additional axioms express this property:

$$\texttt{lift}_2\ f\ o_1\ o_2 = \texttt{lift}_2\ (\lambda x\ y.\ f\ y\ x)\ o_2\ o_1 \qquad \text{(*Ob1)}$$

$$\texttt{lift}_2\ f\ o\ o = \texttt{fmap}\ (\lambda x.\ f\ x\ x)\ o \qquad \text{(*Ob2)}$$

$$\texttt{fmap}\ (\texttt{const}\ x)\ o = \texttt{return}\ x \qquad \text{(*Ob3)}$$

The first axiom says that the order of evaluation does not matter for observables. The second says that observables yield the same value each time they are read (at the same point in time). Finally, the third axiom says that observables the values of which are not used can be omitted.

Note that (*Ob2) is equivalent to $\texttt{lift}_2\ (,)\ o\ o = \texttt{fmap}\ (\lambda x.\ (x, x))\ o$ where $(,) = \lambda x\ y.\ (x, y)$ as is easily seen.

The axiom (*Ob3) can be equivalently stated in terms of "$\gg=$" as

$$o \gg= \texttt{const}\ p = p \qquad \text{(Ob3')}$$

as is easily seen from the definition of "$\gg=$", or even using $\texttt{lift}_2$ as

$$\texttt{lift}_2\ \texttt{const} = \texttt{const}. \qquad \text{(Ob3")}$$

One receives generalizations to arbitrary arities for lifts:

Lemma 2.6 ((*Ob1)–(*Ob3) for any arity)**.**

1. *Let* $f :: a_1 \to \ldots \to a_n \to a$ *and* $o_i :: a_i$ *for* $i = 1, \ldots, n$. *Let* $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ *be a permutation defined in the meta language. Define* $f_\pi = \lambda x_1\ \ldots\ x_n.\ f\ x_{\pi(1)}\ \ldots\ x_{\pi(n)}$. *Then*

$$\texttt{lift}_n\ f\ o_1\ \ldots\ o_n = \texttt{lift}_n\ f_\pi\ o_{\pi^{-1}(1)}\ \ldots\ o_{\pi^{-1}(n)}.$$

   In other words:

$$\texttt{lift}_n\ f = (\texttt{lift}_n\ f_\pi)_{\pi^{-1}}$$

2. *If* $n \geq 1$, $f :: a \to \ldots \to a \to b$ *and* $o ::$ `Obs` a, *then*

$$\texttt{lift}_n\ f\ o\ \ldots\ o = \texttt{fmap}\ (\lambda x.\ f\ x\ \ldots\ x)\ o$$

   where n *repetitions are meant by "…".*

3. *For* $n \geq 0$ *define*

$$\begin{aligned}\texttt{const}_n &:: \ a \to a_1 \to \ldots \to a_n \to a\\ \texttt{const}_n &:= \lambda x\ x_1\ \ldots\ x_n.\ x.\end{aligned}$$

   Then

$$\texttt{lift}_n\ (\texttt{const}_n\ x) = \texttt{const}_n\ (\texttt{return}\ x).$$

[14] Or, "they have no side effects".

[15] The equality is meant semantically. If observables are seen as components of a software, reading twice vs. caching the result of a computation might give a considerable difference in performance and the compiler is free to use these laws to optimize for performance.

From (*Ob1)–(*Ob3) one also receives a generalization of (*Mo2):

Lemma 2.7 ((*Mo2) for higher arities)**.**

1. *If* $n \geq 1$, $g :: a_1 \to \ldots \to a_n \to \texttt{Obs}\ a$ *and* $o_i :: \texttt{Obs}\ a_i$ *for* $i = 1, \ldots, n$, *then*

$$\begin{aligned}&\texttt{join}\ (\texttt{lift}_n\ g\ o_1\ \ldots\ o_n)\\ &\quad = \texttt{join}\ (\texttt{lift}_{n-1}\ (\lambda x_2\ \ldots\ x_n.\ o_1 \gg\!= \lambda x_1.\ g\ x_1\ \ldots\ x_n)\ o_2\ \ldots\ o_n).\end{aligned}$$

2. *If* $n \geq 0$ *and* $f :: a_1 \to \ldots \to a_n \to a$, *then*

$$\texttt{lift}_n\ f \circ^n \texttt{join}\ = \texttt{join} \circ_n (\texttt{lift}_n\ (\texttt{lift}_n\ f)).$$

3. *If* $f_i :: b_i \to \texttt{Obs}\ a_i$ *and* $o_i :: \texttt{Obs}\ b_i$ *for* $i = 1, \ldots, n$ *and* f *is as in part 2, then*

$$\texttt{lift}_n\ f\ (o_1 \gg\!= f_1)\ \ldots\ (o_n \gg\!= f_n) = \texttt{join}\ (\texttt{lift}_n\ g\ o_1\ \ldots\ o_n)$$

where $g := \lambda x_1\ \ldots\ x_n.\ \texttt{lift}_n\ f\ (f_1\ x_1)\ \ldots\ (f_n\ x_n)$.

Remark 2.8. Lemma 2.7 is wrong for general monads without (*Ob1)–(*Ob3): Using lemma 2.2, it is easy to see that e.g. the LHS in 2.7.3 builds a chain of "$\gg\!=$" where the o_i appear first in a row and then the values of the f_i while the RHS yields a chain where they are interleaved. For these to be the same, the ordering in "$\gg\!=$" chains must not matter, which is achieved through (*Ob1).

Corollary 2.9. *By combining the lemmas 2.2 and 2.7 one receives for* f, f_i, o_i *as in 2.7.3:*

$$\begin{aligned}&\texttt{lift}_n\ f\ (o_1 \gg\!= f_1)\ \ldots\ (o_n \gg\!= f_n)\\ &\quad = o_1 \gg\!= \lambda x_1.\ \ldots o_n \gg\!= \lambda x_n.\ \texttt{lift}_n\ f\ (f_1\ x_1)\ \ldots\ (f_n\ x_n)\end{aligned}$$

Note how this corollary can be applied several times to collapse multiple levels of lifts and "$\gg\!=$" chains. One can also set some of the f_i to `return` if there is no further "$\gg\!=$".

Remark 2.10. As in remark 2.5, by unrolling the definition of $\texttt{lift}_n$, one receives that one can always

- reduce away multiple occurrences of an observable as arguments to a lift to a single occurrence and
- if the lifted function is constant in a parameter, eliminate that parameter and the associated observable.

Via lemma 2.2, one can apply any lift-collapse rules also for lift-like chains of "$\gg\!=$". For example:

$$\begin{aligned}&o_1 \gg\!= \lambda x_1.\ o_2 \gg\!= \lambda x_2.\ f\ x_1\ x_2\\ &= \texttt{join}\ (\texttt{lift}_2\ f\ o_1\ o_2)\\ &= \texttt{join}\ \left(\texttt{lift}_2 f_{(1\ 2)} o_2 o_1\right)\\ &= o_2 \gg\!= \lambda x_2.\ o_1 \gg\!= \lambda x_1.\ \ f\ x_1\ x_2\end{aligned}$$

The essential idea is that as far as only lifts are concerned, one may freely reorder lift arguments and instances of "$\gg\!=$" as long as the dependencies between the variables are not violated, i.e. as long as the expression stays only *syntactically* valid. This will change in section 2.3 when the new combinators `ever` and `always` are introduced.

Notation 2.11. For functional operator symbols, I leave out the "$\texttt{lift}_2$" from now on. For example, I just write $o_1 + o_2$ to mean $\texttt{lift}_2\ (+)\ o_1\ o_2$ when it is clear from the context that o_1 and o_2 are observables of numeric type. Also for the unary operator "$\neg$", I write $\neg b$ for $\texttt{fmap}\ (\neg)\ b$.

I write non-`Obs` values in a `Obs` expression to mean their `return`. For example, if $x :: \mathbb{R}$ and $o :: \texttt{Obs}\ \mathbb{R}$, I write $x + o$ for $(\texttt{return}\ x) + o$ for $\texttt{lift}_2\ (+)\ (\texttt{return}\ x)\ o$.

2.2 Boolean observables as market conditions

`Obs Bool` has the special role of describing time periods or probabilistic sets of points in time (or just *time ranges*), identifying $b :: \texttt{Obs Bool}$ with the set of moments where it is true, but in a notion much more restricted than set theory. For example, if $S :: \texttt{Obs}\ \mathbb{R}^+$ describes a share price, then $\texttt{lift}_2\ (\leq)\ S\ (\texttt{return}\ 200)$ – for which I write short "$S \leq 200$" – describes the market condition that S is below or equal to 200 at a given point in time.

I first introduce the reasonable short notation for boolean operations:

Notation 2.12. If $R :: \mathscr{R}\,(a_1 \times \ldots \times a_n)$ is a relational symbol and $\hat{R} :: a_1 \to \ldots \to a_n \to \texttt{Bool}$ is its functional lift, write just R for $\hat{R}$. In particular, write $\texttt{lift}_n\ R$ for $\texttt{lift}_n\ \hat{R}$. Cf. section A.1.4 for details about functional lifts. Recap that any relational symbol has a functional lift except for equality on `Obs` a and `Con` and "$\preceq_.$" from section 3.

As in notation 2.11, I leave out the "$\texttt{lift}_2$" for binary operators if it is clear which objects are observables and which are not. For example, if $o_1, o_2 :: \texttt{Obs}\ \mathbb{R}$, I will write $o_1 \leq o_2$ for $\texttt{lift}_2\ (\leq)\ o_1\ o_2$ for $\texttt{lift}_2\ (\hat{\leq})\ o_1\ o_2$. And if $b_1, b_2 :: \texttt{Obs Bool}$, I write $b_1 \to b_2$ for $\texttt{lift}_2\ (\to)\ b_1\ b_2$.

Also as in notation 2.11, if $x :: \mathbb{R}$ and $o :: \texttt{Obs}\ \mathbb{R}$, I just write e.g. $x \leq o$ to mean $\texttt{lift}_2\ (\leq)\ (\texttt{return}\ x)\ o$.

Theorem 2.14 will show that the notation $o_1 = o_2$ is not misleading.

`Obs Bool` defines a non-standard logic if one introduces the following notation:

Notation 2.13. For the sake of brevity, I write `OB` for `Obs Bool`. Define $\top = \texttt{return True}$ and $\bot = \texttt{return False}$.

For $b, c :: \texttt{OB}$ write

$$b \Rightarrow c$$

iff $(b \to c) = \top$.

The following theorems will witness that one receives the expected laws for these and that hence arguing at the level of observables can be naturally done.

It makes sense to require that the lattice of boolean observables be nontrivial:

$$\top \neq \bot \tag{*2.1}$$

The assumption of this axiom will be used without further reference as a technical simplification. – One could do without it.

Theorem 2.14. *Let $o, p :: \texttt{Obs}\ a$ such that equality is lifted for a. Then*

$$\texttt{lift}_2\ (=)\ o\ p = \top \ \Leftrightarrow\ o = p.$$

Proof. "⇐": By (*Ob2) and (*Ob3), $\texttt{lift}_2\ (=)\ o\ o = \texttt{fmap}\ (\lambda x.\ (=)\ x\ x)\ o = \texttt{fmap}\ (\texttt{const True})\ o = \texttt{return True} = \top$.

"⇒": I show that both o and p are equal to

$$q := \texttt{lift}_3\ \texttt{if}'\ (\texttt{lift}_2\ (=)\ o\ p)\ o\ p$$

where $\texttt{if}' :: \texttt{Bool} \to a \to a \to a$ is the choice function. For $o = q$, note that, by $\texttt{lift}_2\ (=)\ o\ p = \texttt{return True}$,

$$\begin{aligned} q &= \texttt{lift}_3\ \texttt{if}'\ (\texttt{return True})\ o\ p \\ &= \texttt{lift}_2\ (\lambda x\ y.\ \texttt{if}'\ \texttt{True}\ x\ y)\ o\ p \\ &= \texttt{lift}_1\ \texttt{id}\ o = \texttt{fmap id}\ o = o \end{aligned}$$

where the second equality follows from lemma 2.4 and remark 2.5 and the third follows from lemma 2.6.3 as $\lambda x\ y.\ \texttt{if}'\ \texttt{True}\ x\ y = \lambda x\ y.\ x = \lambda x.\ \texttt{const}\ x$.

For $p = q$, apply lemma 2.3 to the definition of q to receive

$$q = \texttt{lift}_4\ (\lambda x_1\ y_1\ x_2\ y_2.\ \texttt{if}'\ (x_1 = y_1)\ x_2\ y_2)\ o\ p\ o\ p.$$

By lemma 2.6.2 applied twice, this is equal to

$$\texttt{lift}_2\ (\lambda x\ y.\ \texttt{if}'\ (x = y)\ x\ y)\ o\ p.$$

The lifted expression is constant in x and reduces to $\lambda y.\ y = \texttt{id}$, hence, by lemma 2.6.3, $q = \texttt{lift}_1\ \texttt{id}\ p = \texttt{fmap id}\ p = p$. □

Note how for $b, c :: \texttt{OB}$ we have $b \Leftrightarrow c$ iff $b = c$ now: "↔" is just equality on `Bool`.

Remark 2.15. The previous theorem did not use axiom (*2.1). It is easy to see the following dichotomy:

- Using axiom (*2.1), it follows that $\texttt{return} :: a \to \texttt{Obs}\ a$ is injective for any type a for which equality is lifted.
- If axiom (*2.1) was false, i.e. $\top = \bot$, then there would be only a single value of type $\texttt{Obs}\ a$ for any type a for which equality is lifted.

Lemma 2.16 (Modus ponens and chain rule for $\texttt{Obs Bool}$)**.** *Let $b, c, d :: \texttt{OB}$.*

1. *If $b \Rightarrow c$ and $b = \top$, then $c = \top$.*
2. *If $b \Rightarrow c$ and $c \Rightarrow d$, then $b \Rightarrow d$.*

Proof. First show 1. We have, by assumption and lemma 2.4/remark 2.5

$$\begin{aligned} \top &= \texttt{lift}_2\ (\to)\ b\ c \\ &= \texttt{lift}_2\ (\to)\ (\texttt{return True})\ c \\ &= \texttt{fmap}\ (\lambda x.\ \texttt{True} \to x)\ c \\ &= \texttt{fmap id}\ c = c \end{aligned}$$

where the last line follows because $(\lambda\, x.\ \texttt{True} \to x) = \texttt{id}$.

For 2, consider the observable

$$q = (b \to c) \to ((c \to d) \to (b \to c))$$

By the lift-collapsing lemma 2.3, $q = \texttt{lift}_3\ f\ b\ c\ d$ where

$$\begin{aligned} f &= \lambda\, x\ y\ z.\ (x \to y) \to ((y \to z) \to (x \to z)) \\ &= \texttt{const}_3\ \texttt{True} \end{aligned}$$

By lemma 2.6.3, $q = \top$. Now, by 1, $(c \to d) \to (b \to c) = \top$. And by applying it again, one receives the desired result $b \to c = \top$. □

The following lemma will provide some notion of quantification to the `OB` framework:

Lemma 2.17. *Let a be a type such that equality is lifted for a. Let $o :: \texttt{Obs}\ a$.*

1. *Let b be another type with lifted equality, $p :: \texttt{Obs}\ a$ and $f :: a \to \texttt{Obs}\ b$. Then $(o = p) \Rightarrow ((o \gg\!= f) = (p \gg\!= f))$*
2. *If $\phi :: a \to \texttt{OB}$, then $(o \gg\!= \phi) = \top$ iff $\forall x :: a : (o = x) \Rightarrow \phi\ x$.*
3. *If $\phi, \psi :: a \to \texttt{OB}$, then $(o \gg\!= \phi) \Rightarrow (o \gg\!= \psi)$ iff $\forall x :: a : (o = x) \Rightarrow (\phi\ x \to \phi\ x)$.*

Proof. 1: One needs to show that

$$((\texttt{lift}_2\ (=)\ o\ p) \to (\texttt{lift}_2\ (=)\ (o \gg\!= f)\ (p \gg\!= f))) = \top.$$

Use corollary 2.9 to see that the conclusion is equal to

$$o \gg\!= \lambda\, x_2.\ p \gg\!= \lambda\, y_2.\ \texttt{lift}_2\ (=)\ (f\ x_2)\ (f y_2).$$

The premise is by definition

$$o \gg\!= \lambda\, x_1.\ p \gg\!= \lambda\, y_1.\ \texttt{return}\ (x_1 = y_1).$$

By applying corollary 2.9 twice, the whole statement is equal to

$$o \gg\!= \lambda\, x_1.\ p \gg\!= \lambda\, y_1.\ o \gg\!= \lambda\, x_2.\ p \gg\!= \lambda\, y_2.\ (x_1 = y_1) \to (f\ x_2 = f\ y_2).$$

By bind-reordering/collapse (remark 2.10), this is equal to

$$o \gg\!= \lambda\, x.\ p \gg\!= \lambda\, y.\ (x = y) \to (f\ x = f\ y).$$

The inner lambda term is $\texttt{const}_2\ \top$: For any x and y, if $x \neq y$, the premise is trivially $\bot$, and for $x = y$, the conclusion is trivially $\top$. Hence, the whole expression is $\top$.

2: If $(o \gg\!= \phi) = \top$, let $x :: a$ and apply part 1 to $p = \texttt{return}\ x$ and $f = \phi$ to receive

$$\begin{aligned} \top &= (o = \texttt{return}\ x) \to ((o \gg\!= \phi) = (\texttt{return}\ x \gg\!= \phi)) \\ &= (o = x) \to ((o \gg\!= \phi) = (\phi\ x)) \\ &= (o = x) \to (\top = (\phi\ x)) \\ &= (o = x) \to \phi\ x \end{aligned}$$

where the second step is by assumption and the last step is because $(\top = \phi\ x)$ $= \texttt{lift}_2\ (=)\ (\texttt{return True})\ (\phi\ x) = \texttt{fmap}\ (\lambda z.\ (\texttt{True} = z))\ (\phi\ x) = \texttt{fmap id}\ (\phi\ x)$ $= \phi\ x$.

For the other direction, assume that $\forall x :: a : o = x \Rightarrow \phi\ x$. Then

$$(o \gg\!= \lambda x.\ o = x \to \phi\ x) = \top$$

because the inner lambda is `const` $\top$. The LHS here in turn is equal to

$$\begin{aligned} & o \gg\!= \lambda x.\ o \gg\!= \lambda y.\ (y = x) \to \phi\ x \\ = {} & o \gg\!= \lambda x.\ (x = x) \to \phi\ x \\ = {} & o \gg\!= \lambda x.\ \phi\ x \\ = {} & o \gg\!= \phi \end{aligned}$$

3: Apply part 2 to $\phi' := \lambda x.\ \phi\ x \to \psi\ x$ and note that

$$(o \gg\!= (\lambda x.\ \phi\ x \to \psi\ x)) = (o \gg\!= \phi) \to (o \gg\!= \psi)$$

by a lift collapsing technique similar to above. □

Remark 2.18. If equality is not lifted for a, one still receives weaker forms for the parts 2.17.2 and 2.17.3, but without connection to the observable o:

If $\phi\ x = \top$ for all x, then $o \gg\!= \phi = \top$ (a direct consequence of (Ob3')). If $\phi\ x \Rightarrow \psi\ x$ for all x, then $(o \gg\!= \phi) \Rightarrow (o \gg\!= \psi)$.

2.3 Quantifying over time

With the combinators introduced so far, observables can only be combined at the same points in time: In an expression consisting only of the `Obs`-combinators "$\gg\!=$" and `return` and ordinary functions, a variable bound by "$\gg\!=$" can only be applied to another function and then, ultimately, to `return`. Hence, any such expression can be reduced to a lift.

Observables should be able to look at previous points in time[16] as well, e.g. to detect whether a barrier on the price of a share has been passed. Such a feature is added by the two new combinators in figure 4 together with requiring that the following boolean observables should be equal to $\top$ for any $b, c :: \texttt{OB}$:

$$\mathfrak{a}\ (b \to c) \to (\mathfrak{a}\ b \to \mathfrak{a}\ c) \qquad \text{(*K)}$$

$$\mathfrak{e}\ b \leftrightarrow \neg \mathfrak{a}\ \neg b \qquad \text{(*Dual)}$$

$$\mathfrak{a}\ \top \qquad \text{(*Gen)}$$

$$\mathfrak{e}\ (\mathfrak{e}\ b) \to \mathfrak{e}\ b \qquad \text{(*4)}$$

$$b \to \mathfrak{e}\ b \qquad \text{(*T)}$$

$$\mathfrak{e}\ b \wedge \mathfrak{e}\ c \to (\mathfrak{e}\ (b \wedge \mathfrak{e}\ c) \vee \mathfrak{e}\ (\mathfrak{e}\ b \wedge c)) \qquad \text{(*.3)}$$

A strong connection to modal logic[17] can be seen here: If one replaces $\mathfrak{a}$ with $\Box$ and $\mathfrak{e}$ with $\Diamond$, then (*K), (*Dual) and (*Gen) are exactly the axioms for a normal modal logic, i.e. one that is realized through Kripke frames, and axioms (*4), (*T), and (*.3) axiomatize transitivity, reflexivity and linearity into the

[16] and *not* at the future, of course.

[17] All definitions and results mentioned in this paragraphs can be found in [8, p. 189 ff.].

Figure 4 Primitives for quantification over time

`ever` :: `OB` → `OB`

`ever` b is `True` in situations where b has ever been true before (including when it is true right now). I also write $\mathfrak{e}$ for `ever`.

`always` :: `OB` → `OB`

`always` b is `True` if b has always been true until the current point in time. In particular, then b is true right now. I also write $\mathfrak{a}$ for `always`.

Figure 5 Some laws for $\mathfrak{e}/\mathfrak{a}$

1. $\mathfrak{e}\ (\mathfrak{e}\ b) = \mathfrak{e}\ b$.
   $\mathfrak{a}\ (\mathfrak{a}\ b) = \mathfrak{a}\ b$.
2. $\mathfrak{a}\ b \Rightarrow b \Rightarrow \mathfrak{e}\ b$.
3. If $b \Rightarrow \mathfrak{e}\ c$, then $\mathfrak{e}\ b \Rightarrow \mathfrak{e}\ c$.
   If $\mathfrak{a}\ c \Rightarrow b$, then $\mathfrak{a}\ c \Rightarrow \mathfrak{a}\ b$.
4. If $b \Rightarrow c$, then $\mathfrak{e}\ b \Rightarrow \mathfrak{e}\ c$ and $\mathfrak{a}\ b \Rightarrow \mathfrak{a}\ c$.
5. $\mathfrak{e}\ \bot = \mathfrak{a}\ \bot = \bot$.
   $\mathfrak{e}\ \top = \mathfrak{a}\ \top = \top$.
6. $\mathfrak{e}\ b \wedge \mathfrak{a}\ c \Rightarrow \mathfrak{e}\ (b \wedge c)$
7. $\mathfrak{e}\ (b \vee c) = \mathfrak{e}\ b \vee \mathfrak{e}\ c$.
   $\mathfrak{a}\ (b \wedge c) = \mathfrak{a}\ b \wedge \mathfrak{a}\ c$.
8. $\mathfrak{e}\ (b \wedge c) \Rightarrow \mathfrak{e}\ b \wedge \mathfrak{e}\ c$.
   $\mathfrak{a}\ (b \vee c) \Leftarrow \mathfrak{a}\ b \vee \mathfrak{a}\ c$.
9. $\mathfrak{a}\ (b \to \mathfrak{e}\ c) \Rightarrow \mathfrak{a}\ (\mathfrak{e}\ b \to \mathfrak{e}\ c)$.

past[18] of the visibility relation "past states of the world", respectively. The resulting modal logic is called *S4.3*

The semantic consequences are easy to see for (*4) and (*T). For the intuitive understanding of (*.3), consider a point in time such that there are two previous points in time where b was true at one and c was true at the other. Then one of them has to be later and, hence, see the other as a past point in time again.[19]

It is easy to see that any statement provable from S4.3 also holds in $\mathrm{LPT}_{\mathrm{Obs}}$ in the following sense: Let ϕ be a statement in the propositional modal language with countably many propositional letters and let $p_1, \ldots, p_n$ be the propositional variables in ϕ. Let ϕ' be the term in the language of observables with n free variables $b_1, \ldots, b_n$ arising from ϕ by replacing p_i with b_i, $\Box$ with $\mathfrak{a}$ and $\Diamond$ with $\mathfrak{e}$. If ϕ is a theorem of S4.3, then $\forall b_1 \ldots \forall b_n : \phi'$ is a theorem of $\mathrm{LPT}_{\mathrm{Obs}}$.

The following lemma provides a handy collection of such theorems.

Lemma 2.19. *Whenever b, c :: `OB`, the statements from figure 5 hold.*

Proof. I only show one of the versions for $\mathfrak{e}$ or $\mathfrak{a}$ for each point. The respective other versions can be received by applying (*Dual).

1: The $\mathfrak{e}$ variant follows directly from (*4) and (*T).

[18] (*.3) does *not* state that the visibility relation be linear: Any two *past* points in time must be related, but two *future* states may be unrelated. The canonical model where one holds, but the other does not is where the states of the world are the nodes of a tree.

[19] In [8], the RHS of (*.3) contains the additional case $\mathfrak{e}\ (b \wedge c)$ (translating $\Diamond$ back to $\mathfrak{e}$) for the case where the two points in time are equal. It is clear that this can be left out when (*T) is assumed.

2 is just (*T) and its converse by negation.

3: If $\mathfrak{a}\ b \Rightarrow c$, then by (*K) and modus ponens, $\mathfrak{a}\ (\mathfrak{a}\ b) \Rightarrow \mathfrak{a}\ c$ and by 1, $\mathfrak{a}\ (\mathfrak{a}\ b) = \mathfrak{a}\ b$.

4 is just (*K) (and modus ponens).

5: I show the first line. $\mathfrak{a}\ \bot \Rightarrow \bot$ by 2, so $\mathfrak{a}\ \bot = \bot$. $\mathfrak{e}\ \bot = \bot$ is the converse by negation of (*Gen).

6: Apply (*K) to b and $\neg c$ in place of c to receive that the following is equal to $\top$:

$$\mathfrak{a}\ (b \to \neg c) \to (\mathfrak{a}\ b \to \mathfrak{a}\ \neg c)$$

The converse of this is

$$\neg(\mathfrak{a}\ b \to \mathfrak{a}\ \neg c) \to \neg(\mathfrak{a}\ (b \to \neg c))$$

or, in other words,

$$(\mathfrak{a}\ b \wedge \mathfrak{e}\ c) \to \mathfrak{e}\ (b \wedge c)$$

the RHS of which implies $\mathfrak{e}\ c$ because $b \wedge c \Rightarrow c$.

7: "$\Leftarrow$": $b \Rightarrow b \vee c$, so by 4 $\mathfrak{e}\ b \Rightarrow \mathfrak{e}\ (b \vee c)$. Analogously, $\mathfrak{e}\ c \Rightarrow \mathfrak{e}\ (b \vee c)$. Hence, $\mathfrak{e}\ b \vee \mathfrak{e}\ c \Rightarrow \mathfrak{e}\ (b \vee c)$.

"$\Rightarrow$": Consider the negation:

$$\begin{aligned} &\mathfrak{e}\ (b \vee c) \wedge \neg(\mathfrak{e}\ b \vee \mathfrak{e}\ c) \\ \Leftrightarrow\ &\mathfrak{e}\ (b \vee c) \wedge \mathfrak{a}\ (\neg b) \wedge \mathfrak{a}\ (\neg c) \end{aligned}$$

By 6 applied twice, this implies

$$\mathfrak{e}\ ((b \vee c) \wedge \neg b \wedge \neg c) = \mathfrak{e}\ \bot = \bot$$

where the last equality is due to 5.

8 follows just like the "$\Leftarrow$" case of 7.

9: By 3, it suffices to show:

$$\mathfrak{a}\ (b \to \mathfrak{e}\ c) \Rightarrow (\mathfrak{e}\ b \to \mathfrak{e}\ c)$$

By the previous parts, we have

$$\begin{aligned} &\mathfrak{a}\ (b \to \mathfrak{e}\ c) \wedge \mathfrak{e}\ b \\ \Rightarrow\ &\mathfrak{e}\ ((b \to \mathfrak{e}\ c) \wedge b) \\ \Rightarrow\ &\mathfrak{e}\ (\mathfrak{e}\ c) = \mathfrak{e}\ c. \end{aligned}$$

□

Note that the proof of the previous lemma did not use axiom (*.3), so it would also be valid in the weaker normal logic S4 which misses that axiom. The following requires axiom (*.3):

Lemma 2.20.

$$\mathfrak{e}\ \mathfrak{a}\ b \Rightarrow \mathfrak{a}\ \mathfrak{e}\ b \tag{G}$$

Proof. The negation of $\mathfrak{e}\ \mathfrak{a}\ b \to \mathfrak{a}\ \mathfrak{e}\ b$ is

$$\begin{aligned} &\mathfrak{e}\ \mathfrak{a}\ b \wedge \neg\mathfrak{a}\ \mathfrak{e}\ b \\ \Leftrightarrow\ &\mathfrak{e}\ \mathfrak{a}\ b \wedge \mathfrak{e}\ \mathfrak{a}\ \neg b. \end{aligned}$$

Application of axiom (*.3) yields

$$\mathfrak{e}\ (\mathfrak{a}\ b \wedge \mathfrak{e}\ \mathfrak{a}\ \neg b) \vee \mathfrak{e}\ (\mathfrak{e}\ \mathfrak{a}\ b \wedge \mathfrak{a}\ \neg b).$$

By the laws from lemma 2.19,

$$\begin{aligned} & \mathfrak{a}\ b \wedge \mathfrak{e}\ \mathfrak{a}\ \neg b \\ \Rightarrow\ & \mathfrak{a}\ b \wedge \mathfrak{e}\ \neg b \\ \Rightarrow\ & \mathfrak{e}\ (b \wedge \neg b) = \mathfrak{e}\ \bot = \bot. \end{aligned}$$

Similarly, one receives $\mathfrak{e}\ \mathfrak{a}\ b \wedge \mathfrak{a}\ \neg b = \bot$. Then the whole expression above is $\bot$. □

Example 2.21. Note how one does *not* in general receive any "lift collapsing" lemmas for $\mathfrak{e}/\mathfrak{a}$. For example, if $o :: \texttt{Obs}\ \mathbb{R}^+$ is e.g. a share price, then

$$\begin{aligned} & o \gg\!= \lambda x.\ \mathfrak{e}\ (o \gg\!= \lambda y.\ (y > x)) \\ \neq\ & o \gg\!= \lambda x.\ o \gg\!= \lambda y.\ \mathfrak{e}\ (y > x). \end{aligned}$$

The first expression states that o is at a (not in general unique) maximum of o since the beginning of time. The second is simply $\bot$ because it can be (lift-) reduced to

$$o \gg\!= \lambda x.\ \mathfrak{e}\ (x > x)$$

and for any x, $x > x = \bot$, so $\mathfrak{e}\ (x > x) = \bot$ by lemma 2.19.5.

2.4 Defining time

Notation 2.22. For the sake of brevity, define:

$$\mathfrak{n} := \texttt{now}$$

The following axioms state that the `now` observable basically reflects the `Time` type. They are easily verified by intuition. Let $t :: \texttt{Time}$ and $b :: \texttt{OB}$. Then

$$\mathfrak{e}\ (\mathfrak{n} = t) = \mathfrak{n} \geq t \tag{*2.2}$$

$$\mathfrak{e}\ (\mathfrak{n} = t \wedge b) \Rightarrow \mathfrak{a}\ (\mathfrak{n} = t \rightarrow b) \tag{*2.3}$$

$$(\mathfrak{n} \gg\!= \lambda t.\ \mathfrak{a}\ (\mathfrak{n} \leq t)) = \top. \tag{*2.4}$$

The "⇒" direction of axiom (*2.2) states that `now` is monotonically increasing. To see that intuitively, let t be some previous value of `now`. Then $\mathfrak{e}\ (\texttt{now} = t)$ is true, hence the current value of `now` is $\geq t$. The other direction states that any previous point in time as of the `Time` type did actually exist.

Axiom (*2.3) basically states that `now` is *strictly* increasing in time: Any value of `now` fixes all possible conditions of type `OB`: Nothing may change while the value of `now` stays the same. Another point of view is that if time is discrete, then `now` must have the highest granularity.

(*2.4) is essentially a monadic variant of (*2.2). It has to be stated here explicitly due to formal restrictions.

Remark 2.23. The "⇐" direction of axiom (*2.2) is not actually used in the following, but simplifies some arguments. One can always restrict the `Time` type accordingly. The `timeOffset` function is flexible enough so that this does not cause any problems.

Intuitively, (*2.4) should follow from (*2.2). However, the framework is not yet able to support the required pattern of argumentation (cf. section 6.1).

2.4.1 Earlier and first occurrences of an event

Define the following functions:

$$\begin{aligned}\texttt{earlier} &:: \texttt{OB} \to \texttt{OB}\\ \texttt{earlier}\ b &= \mathfrak{n} \gg\!= \lambda t.\ \mathfrak{e}\ (\mathfrak{n} < t \wedge b)\end{aligned}$$

$$\begin{aligned}\texttt{first} &:: \texttt{OB} \to \texttt{OB}\\ \texttt{first}\ b &= b \wedge \neg\texttt{earlier}\ b\end{aligned}$$

Also write $\bar{\mathfrak{e}}$ for `earlier` and $\mathfrak{f}$ for `first`.

Example 2.24. Let $o :: \texttt{Obs}\ \mathbb{R}$. Then the following boolean observable is `True` when and only when o is at a strict all-time high:

$$o \gg\!= \lambda x.\ \neg\bar{\mathfrak{e}}\ (o \geq x)$$

$\bar{\mathfrak{e}}\ b$ is `True` iff b has happened strictly before the current point in time, i.e. if $\mathfrak{e}\ b$ is true, but it is not true for the first time. Indeed, it is easily seen that $\bar{\mathfrak{e}}\ b = \mathfrak{e}\ b \wedge \neg\mathfrak{f}\ b$ using the following lemma.

Lemma 2.25.

$$\mathfrak{e}\ (b \wedge c) \wedge \neg c \Rightarrow \bar{\mathfrak{e}}\ b$$

Proof. I show that

$$\neg\bar{\mathfrak{e}}\ b \wedge \mathfrak{e}\ (b \wedge c) \Rightarrow c.$$

To see this, first note that using axiom (*2.4), lift reduction and the properties of $\mathfrak{e}/\mathfrak{a}$:

$$\begin{aligned}\neg\bar{\mathfrak{e}}\ b &= \neg\bar{\mathfrak{e}}\ b \wedge (\mathfrak{n} \gg\!= \lambda t.\ \mathfrak{a}\ (\mathfrak{n} \leq t))\\ &= \mathfrak{n} \gg\!= \lambda t.\ \neg\mathfrak{e}\ (\mathfrak{n} < t \wedge b) \wedge \mathfrak{a}\ (\mathfrak{n} \leq t)\\ &= \mathfrak{n} \gg\!= \lambda t.\ \mathfrak{a}\ (\mathfrak{n} \geq t \vee \neg b) \wedge \mathfrak{a}\ (\mathfrak{n} \leq t)\\ &= \mathfrak{n} \gg\!= \lambda t.\ \mathfrak{a}\ ((\mathfrak{n} \geq t \vee \neg b) \wedge \mathfrak{n} \leq t)\\ &= \mathfrak{n} \gg\!= \lambda t.\ \mathfrak{a}\ ((b \to \mathfrak{n} \geq t) \wedge \mathfrak{n} \leq t)\\ &\Rightarrow \mathfrak{n} \gg\!= \lambda t.\ \mathfrak{a}\ (b \to \mathfrak{n} = t).\end{aligned}$$

Now, applying lemma 2.19.6:

$$\begin{aligned}&\mathfrak{e}\ (b \wedge c) \wedge \neg\bar{\mathfrak{e}}\ b\\ \Rightarrow\ &\mathfrak{n} \gg\!= \lambda t.\ \mathfrak{e}\ (b \wedge c) \wedge \mathfrak{a}\ (b \to \mathfrak{n} = t)\\ \Rightarrow\ &\mathfrak{n} \gg\!= \lambda t.\ \mathfrak{e}\ (\mathfrak{n} = t \wedge c)\end{aligned}$$

By axiom (*2.3), $\mathfrak{e}\ (\mathfrak{n} = t \wedge c) \Rightarrow \mathfrak{a}\ (\mathfrak{n} = t \to c) \Rightarrow (\mathfrak{n} = t \to c)$ for any t. So the above implies

$$\begin{aligned}&\mathfrak{n} \gg\!= \lambda t.\ \mathfrak{n} = t \to c\\ \Rightarrow\ &\mathfrak{n} \gg\!= \lambda t.\ (\mathfrak{n} = t) \wedge (\mathfrak{n} = t \to c)\\ \Rightarrow\ &\mathfrak{n} \gg\!= (\texttt{const}\ c)\\ =\ &c.\end{aligned}$$

□

This concludes the definition of the theory LPT_{Obs}. The following section will define the remaining sorts, symbols and axioms for the theory LPT.

3 Contracts

This section will introduce the basic building blocks of contracts. A contract will be the only way to talk about any kind of financial "asset". The only thing a market participant will be able to do with a contract is acquiring it. For example, in the framework, there is no notion of "buying" a "troy ounce of gold". Instead, any market participant will be able to acquire at any time the contract that

- obliges her to immediately pay the current value of the gold price (a certain observable of type $\mathbb{R}^+$) in (say) USD and
- grants her the right to receive, at any future point in time, the value of the gold price in USD.

We will later be able to state that since anyone can freely acquire it, nobody should be able to make risk-free profit from this contract, i.e. it must be $\preceq 0$.

Introduce first a new type `Con` (short for "contract") together with primitive operations and their intended meanings as in figure 6. Note that while `Obs` was a type *constructor* – there is a different type `Obs` a for any type a – `Con` is a single type. Also introduce a type `Currency` the values of which are to be interpreted as the different currencies available.

In comparison to my approach, [3] lacks a `read′` primitive. This was partly compensated for by introducing `cond` from figure 7 below as a primitive and giving another "until" parameter to `anytime`. My approach is more general as the examples for "$\rightsquigarrow$" below will show.

While contracts are usually made between two parties, the language of contracts presented here only models a single side, namely the "holder" of the contract. This can also be viewed in such a way that the counterparty is a big, anonymous and forgetful entity called "the" stock exchange. The `give` combinator would then just "flip the contract over" to have the holder take the position of the stock exchange.

Remark 3.1.

1. `give` x does not *only* change signs: If x allows the holder to make a choice, the holder of `give` x must be willing to accept any choice a counterparty would make.

2. For `when′`, there is also a more natural combinator `when` of the same type that will wait for the *next* time b becomes true. `when′` was chosen here for its comparatively simple algebraic properties.

3. Also for `when′`, the "first time b becomes true" may not exist. For example, assume that time is continuous and consider $b = (\mathfrak{n} > t)$. However, the following axioms are generally not affected by this issue and in some cases `when′` $(\mathfrak{n} > t)$ x can in fact be given a sensible meaning. Cf. section 3.4 below.

Note how `read′` :: `Obs Con` $\to$ `Con` is similar to `join` :: `Obs` (`Obs` a) $\to$ `Obs` a. Indeed, its semantics are related and we will require similar axioms for it. And

Figure 6 Primitives for contracts

`zero` :: `Con`
: The empty contract, stating no rights or obligations.

`one` :: `Currency` → `Con`
: The contract `one` k immediately pays a single unit of the currency k to the holder of the contract.

  Often, the k argument does not matter as long as it is always the same. In these cases, I will omit it.

`and` :: `Con` → `Con` → `Con`
: Acquiring `and` x y is equivalent to acquiring both x and y. `and` can be seen as a portfolio construction operator.

`give` :: `Con` → `Con`
: Acquiring `give` x means acquiring the counterparty's side in the contract x. This means that all permissions become obligations and vice versa and all payments change signs.

`scale` :: $\mathbb{R}^+$ → `Con` → `Con`
: `scale` α x scales all payments within x by a factor of α.

`or` :: `Con` → `Con` → `Con`
: A market participant acquiring `or` x y must acquire immediately exactly one of x or y (but not both or none).

`when′` :: `OB` → `Con` → `Con`
: `when′` b x obliges the holder to acquire x as soon as $\mathfrak{e}$ b becomes true. I.e. if b has ever been true before acquisition time, x is acquired immediately and otherwise, x is acquired the first time b becomes true.

`anytime` :: `Con` → `Con`
: `anytime` x grants the right (but not the obligation) to acquire x at any time in the future.

`read′` :: `Obs Con` → `Con`
: At the moment a market participant acquires `read′` p, the observable p is read and the holder is obliged to immediately acquire the resulting contract.

similarly to `join`, where one defined "$\gg\!=$", one can define the following helper function:

$$(\rightsquigarrow) :: \texttt{Obs}\ a \to (a \to \texttt{Con}) \to \texttt{Con}$$
$$o \rightsquigarrow f = \texttt{read}'\ (\texttt{fmap}\ f\ o)$$

$o \rightsquigarrow f$ reads, on acquisition, the observable o, applies f to the result and acquires the resulting contract.

Example 3.2. The gold-buying contract above can now be written as follows, given a gold price $g :: \texttt{Obs}\ \mathbb{R}^+$:

$$x := \texttt{and}\ (\texttt{give}\ (\texttt{money}\ g))\ (\texttt{anytime}\ (\texttt{money}\ g))$$
$$\text{where} \quad \texttt{money}\ g := g \rightsquigarrow \lambda\alpha.\ \texttt{scale}\ \alpha\ \texttt{one}$$

`money` g means reading, on acquisition, the gold price g, then receiving as many dollars as that value was. Then x means at the same time paying (the "reverse" of receiving) g and receiving the option to get back g at any future point in time.

It is expected that x is of non-negative value because the holder could choose to exercise the `anytime`-option immediately, receiving 0 in total. Indeed, it will be a simple consequence of the below axioms for `give`, `and` and `anytime` that $x \succeq$ `zero`.

If g is in fact a well-defined "gold price", one would expect that x can be acquired at the marked without any further payments: x is "available at the market" or x "has price 0" – or `anytime` (`money` g) has "price" g. In fact, I will use as the definition of a "price" for `anytime` (`money` g) that $x \approx$ `zero` (definition 4.1).

We will later be able to show that from this fact, it follows that `money` g is preferable to `when'` b (`money` g) in present value: It is always better to receive gold early, a property shared with `one` if non-negative interest rates are assumed (both intuitively and formally). This is a non-trivial property: For example, if g was known to increase sufficiently quickly over time, waiting would be preferable.

Note here that gold also has the special property that it is not possible to do anything with it than selling it later for its market price, a property shared with the concept of a "dividend-free share" introduced below.

As the above examples show, writing contracts using only the primitives is tedious and it is natural to introduce some tools (figure 7): `cond` $b\ x\ y$ acquires x if b is true on acquisition of the compound contract and y otherwise. The `scale`$*$ functions, `money` and `moneyG` are obviously variants of `scale` for different argument types. `when` is the abovementioned more natural variant of `when'` that will always wait for the *next* time b becomes true. `at` and `after` can be used to delay a contract to a specified point in time or for a specified time period. As in general not every `TimeDiff` is valid for every point in time, for example if time is finite, one needs to handle the `Nothing` case of `timeOffset` (cf. section A.3.3).

I use the short notation from figure 8. The axioms and lemmas below will justify this notation.

One can now write the contract x from example 3.2 as

$$(\mathfrak{A}\ g) - g.$$

Figure 7 Tools for building contracts

$$
\begin{aligned}
&\texttt{cond} && :: \texttt{OB} \to \texttt{Con} \to \texttt{Con} \to \texttt{Con} \\
&\texttt{cond}\ b\ x\ y && = b \rightsquigarrow \lambda\beta.\ \texttt{if}'\ \beta\ x\ y \\
&\texttt{scaleG} && :: \mathbb{R} \to \texttt{Con} \to \texttt{Con} \\
&\texttt{scaleG}\ \alpha\ x && = \texttt{if}'\ (\alpha \geq 0)\ (\texttt{scale}\ \alpha\ x)\ (\texttt{give}\ (\texttt{scale}\ (-\alpha)\ )x) \\
&\texttt{scaleO} && :: \texttt{Obs}\ \mathbb{R}^+ \to \texttt{Con} \to \texttt{Con} \\
&\texttt{scaleO}\ o\ x && = o \rightsquigarrow \lambda\alpha.\ \texttt{scale}\ \alpha\ x \\
&\texttt{scaleGO} && :: \texttt{Obs}\ \mathbb{R} \to \texttt{Con} \to \texttt{Con} \\
&\texttt{scaleGO}\ o\ x && = o \rightsquigarrow \lambda\alpha.\ \texttt{scaleG}\ \alpha\ x \\
&\texttt{money} && :: \texttt{Obs}\ \mathbb{R}^+ \to \texttt{Con} \\
&\texttt{money}\ o && = \texttt{scaleO}\ o\ \texttt{one} \\
&\texttt{moneyG} && :: \texttt{Obs}\ \mathbb{R} \to \texttt{Con} \\
&\texttt{moneyG}\ o && = \texttt{scaleGO}\ o\ \texttt{one} \\
&\texttt{when} && :: \texttt{OB} \to \texttt{Con} \to \texttt{Con} \\
&\texttt{when}\ b\ x && = \texttt{now} \rightsquigarrow \lambda t.\ \texttt{when}'\ (b \wedge \texttt{now} \geq t)\ x \\
&\texttt{at} && :: \texttt{Time} \to \texttt{Con} \to \texttt{Con} \\
&\texttt{at}\ t\ x && = \texttt{when}\ (\texttt{now} = t)\ x \\
&\texttt{atMaybe} && :: \texttt{Maybe Time} \to \texttt{Con} \to \texttt{Con} \\
&\texttt{atMaybe}\ (\texttt{Just}\ t)\ x && = \texttt{at}\ t\ x \\
&\texttt{atMaybe Nothing}\ x && = \texttt{zero} \\
&\texttt{after} && :: \texttt{TimeDiff} \to \texttt{Con} \to \texttt{Con} \\
&\texttt{after}\ \Delta t\ x && = \texttt{now} \rightsquigarrow (\lambda t.\ \texttt{atMaybe}\ (\texttt{timeOffset}\ t\ \Delta t)\ x)
\end{aligned}
$$

Figure 8 Short notation for contracts

Let $x, y :: \texttt{Con}$, $\alpha :: \mathbb{R}$, $o :: \texttt{Obs}\ \mathbb{R}$ and $b :: \texttt{OB}$.

For this	also write this
`zero`	0
`one`	1
`give` x	$-x$
`and` x y	$x + y$
`and` x (`give` y)	$x - y$
`or` x y	$x \vee y$
`scaleG` α x	$\alpha \cdot x$
`scaleG` α `one`	α
`scaleGO` o x	$o \cdot x$
`moneyG` o	o
`when'` b x	$\mathfrak{W}'\ b\ x$
`when` b x	$\mathfrak{W}\ b\ x$
`anytime` x	$\mathfrak{A}\ x$

Note that if $\alpha \geq 0$, then $\alpha \cdot x$ can be viewed as short notation for `scale` α x. Analogous for o.

Example 3.3 (A strange contract). As a somewhat more complex example, the following contract z gives the holder the right to receive at any later time the difference between a – say – share price $o :: \texttt{Obs}\ \mathbb{R}^+$ at exertion and at acquisition:

$$z := o \rightsquigarrow \lambda x_0.\ \mathfrak{A}\ (o \rightsquigarrow \lambda x_1.\ x_1 - x_0)$$

z is also called an "American option at the money" (without restrictions on the exertion time here). Cf. section 4.6 and [1, p. 201].

3.1 The present value relation

In order to model the partial order that one contract is "better than another in present value", introduce a single new relational symbol

$$\left(\preceq_. \right) :: \mathscr{R}\left(\texttt{OB}, \texttt{Con}, \texttt{Con}\right),$$

i.e. "$\preceq_.$" is a ternary relation the first argument of which has to be of type `OB` and the other two of type `Con`. If $b :: \texttt{OB}$ and $x, y :: \texttt{Con}$, I write $x \preceq_b y$ to state that "x is less or equal to y in present value under conditions b". Note how the `OB` type is used both to define contracts as in `when'` and to describe market conditions here. The following subsections introduce axioms to make this definition sensible.

Notation 3.4. I follow the usual notational conventions, listed in figure 9.

Define further for $b :: \texttt{OB}$:

$$x \prec_b y \ :\Leftrightarrow\ x \preceq_b y \wedge \forall c :: \texttt{OB},\ c \Rightarrow b,\ c \neq \bot : x \not\succeq_c y$$

Figure 9 Notation for present value relations

Write this	For this
$x \succeq_b y$	$y \preceq_b x$
$x \approx_b y$	$x \preceq_b y$ and $y \preceq_b x$
$x \preceq y$	$x \preceq_\top y$
...	...

$x \prec_b y$ means that b guarantees that x will never be preferable to y. This means that not only $x \not\succeq_b y$, but no matter how one makes the condition b stronger, i.e. more specific, one can never reach a situation where $x \succeq y$.

Note that these do not lift functionally and hence cannot be used to construct observables through lifts: Otherwise, one would be able to define a contract that acquires – say – a contract x as soon as it becomes preferable to a contract y in present value and it is not clear what this is supposed to mean.

The remainder of this section will, based on the intuition from section 1, introduce axioms that should hold for "$\preceq_.$".

3.1.1 Logical axioms

The following axioms allow us to argue about "$\preceq_.$" in a natural way:

First of all, "$\preceq_b$" for fixed b should be a preorder[20] , i.e. the following should hold for all $x, y, z :: \texttt{Con}$ and $b :: \texttt{OB}$:

$$x \preceq_b x \tag{*3.1}$$

$$x \preceq_b y \text{ and } y \preceq_b z \Rightarrow x \preceq_b z \tag{*3.2}$$

Next, "$x \preceq_. y$" for fixed x and y should be compatible with logical deduction on $\texttt{OB}$, expressing the "under conditions" part. The following should hold for $b, c :: \texttt{OB}$ and $x, y :: \texttt{Con}$:

$$(b \Rightarrow c) \text{ and } x \preceq_c y \Rightarrow x \preceq_b y \tag{*3.3}$$

$$x \preceq_b y \text{ and } x \preceq_c y \Rightarrow x \preceq_{b \vee c} y \tag{*3.4}$$

$$x \preceq_\bot y \tag{*3.5}$$

In other words, for any fixed x and y, the formula $x \preceq_b y$ should define an *ideal* in the boolean algebra $\texttt{OB}$.

It is easy to see that now, "$\prec_.$" is transitive and irreflexive and the three latter rules still hold if one replaces "$\preceq$" by "$\prec$". This would not be true for the naive definition of "$x \prec_b y$" as $x \not\succeq_b y$.

Lemma 3.5. *Let $f :: \texttt{Con} \to \texttt{Con}$ be such that for all $b :: \texttt{OB}$ and $x, y :: \texttt{Con}$:*

$$x \preceq_b y \Leftrightarrow f\ x \preceq_b f\ y$$

Then also

$$x \prec_b y \Leftrightarrow f\ x \prec_b f\ y$$

for all x, y and b.

[20] i.e. a partial order where $x \approx_b y$ does not imply $x = y$. In fact, this is usually *wrong* for all $b \neq \top$ and some x and y (but might be true for $b = \top$).

Proof. Define $b_\Rightarrow$ to be the class of $c :: \mathtt{OB}$ such that $c \neq \bot$ and $c \Rightarrow b$.

$$\begin{aligned} x \prec_b y &\Leftrightarrow x \preceq_b y \wedge \forall c \in b_\Rightarrow : \neg(y \preceq_c x) \\ &\Leftrightarrow f\ x \preceq_b f\ y \wedge \forall c \in b_\Rightarrow : \neg(f\ y \preceq_c f\ x) \\ &\Leftrightarrow f\ x \prec_b f\ y \end{aligned}$$

□

Remark 3.6 (Forcing). A certain degree of similarity to the technique of forcing[21] in set theory can be seen here:

If one considers the partial order $(\mathtt{OB}, \Rightarrow)$ without $\bot$ and writes, by heavy abuse of notation, $b \Vdash x \preceq y$ instead of $x \preceq_b y$, then (*3.3) would hold by strengthening of forcing conditions and (*3.4) follows by a simple density argument. If one reads "$\prec$" on the RHS of "$\Vdash$" as »"$\preceq$" and not "$\succeq$"«, then one receives

$$\begin{aligned} b \Vdash x \prec y &\Leftrightarrow b \Vdash (x \preceq y \wedge x \not\succeq y) \\ &\Leftrightarrow (b \Vdash x \preceq y) \wedge (\forall c \Rightarrow b : b \not\Vdash x \succeq y) \end{aligned}$$

where the last equivalence uses a density argument again. Translating back into "$\preceq_b$" notation, the last line is exactly the definition of $x \prec_b y$.

However, most density arguments to *not* work in LPT. For example, whenever forcing a condition is dense in a partial order, already the weakest condition $\top$ forces it. In observables, that would mean that whenever we have that

$$\forall b :: \mathtt{OB}, b \neq \bot : \exists c :: \mathtt{OB}, c \neq \bot, c \Rightarrow b : x \preceq_c y,$$

then $x \preceq y$. It is not clear why this should be.

I will continue to introduce axioms in the order of the primitives as given above, which is supposed to loosely resemble the "complexity" introduced by the combinators.

3.1.2 `zero`, `and`, `give`

The portfolio construction operator `and` should be monotonic and

$$(\mathtt{Con}, \mathtt{zero}, \mathtt{and}, \mathtt{give})$$

should form an abelian group up to "$\approx_b$" for any $b :: \mathtt{OB}$. In detail, I require the following axioms:

`and` should be monotonic for any relation "$\preceq_b$":

$$x_1 \preceq_b y_1 \text{ and } x_2 \preceq_b y_2 \Rightarrow \mathtt{and}\ x_1\ x_2 \preceq_b \mathtt{and}\ y_1\ y_2 \tag{*3.6}$$

Axiom (*3.6) is justified by executing the two arbitrage strategies for x_1 and y_1 and x_2 and y_2, respectively, in parallel.

Next, one requires the abelian group laws from algebra where equality is replaced by "$\approx$":

$$\mathtt{and}\ x\ y \approx \mathtt{and}\ y\ x \tag{*3.7}$$

$$\mathtt{and}\ (\mathtt{and}\ x\ y)\ z \approx \mathtt{and}\ x\ (\mathtt{and}\ y\ z) \tag{*3.8}$$

$$\mathtt{and}\ x\ \mathtt{zero} \approx x \tag{*3.9}$$

$$\mathtt{and}\ x\ (\mathtt{give}\ x) \approx \mathtt{zero} \tag{*3.10}$$

[21] Cf. e.g. [9, chap. 14].

These rules justify writing $(+)$, $(-)$ and 0 for `and`, `give` and `zero`, respectively.

Axioms (*3.7)–(*3.9) are clearly justified from the intuition of `and` and `zero`. Axiom (*3.10) also follows from the intuition of `give`: Acquiring both sides of the same contract must be valued with 0. Note that the contract $x - x =$ `and` x (`give` x) is not automatically risk-free, but only *can be made* risk-free. For example, if $x =$ `anytime` y and a trader acquires $x - x$, then exercises first, she is left with $y -$ `anytime` y which is not in general risk-free.

It is easy to see that `give` is reversely monotonic, i.e. if $x \preceq_b y$, then `give` $x \succeq_b$ `give` y.

Note how "$\approx_b$" can "factor through" the above rules. For example, if it is known that $y \approx_b$ `give` x, then by monotonicity (axiom (*3.6)) also `and` $x\ y \approx_b$ `and` x (`give` x) $\approx$ `zero`, so altogether `and` $x\ y \approx_b$ `zero`.

Now the usual group-theoretic proofs carry to the "$\approx_b$" pseudo groups and one receives for example

$$\texttt{give}\ (\texttt{give}\ x) \approx x$$

as expected.

In a model, one receives abelian groups by forming the equivalence classes with respect to "$\approx_b$". For $b = \bot$, this is the trivial (point) group and if $b \Rightarrow c$, then one receives a projection from the group with respect to "$\approx_c$" to the group with respect to "$\approx_b$".

`give` and `and` are also strictly monotonic in the following sense:

Lemma 3.7. *Let $x_1 \preceq_b y_1$ and $x_2 \prec_b y_2$. Then*

1. `give` $x_2 \succ_b$ `give` y_2.
2. $x_1 + x_2 \prec_b y_1 + y_2$.

Proof. 1: `give` is a self-inverse reversely monotonic map. The statement now follows similarly to lemma 3.5.

2: The map $\lambda x.\ x_1 + x$ is an automorphism of any partial order "$\preceq_c$" with inverse $\lambda x.\ (-x_1) + x$ and hence by lemma 3.5 we have $x_1 + x_2 \prec_b x_1 + y_2$. And $x_1 + y_2 \preceq_b y_1 + y_2$ by monotonicity. □

3.1.3 `one`

The only thing one knows about `one` is that it's always of strictly positive value in the sense of notation 3.4:

$$\texttt{zero} \prec \texttt{one} \tag{*3.11}$$

Intuitively, this means that no currency should be worthless. This assumption is required and reasonable: If a currency ever has literally zero value, it is not clear what prices denoted in this currency are supposed to mean. Vice versa, a core result will be that "$\leq$" on prices behaves like "$\preceq$" on contracts (lemma 3.12 for the static and lemma 3.42 for the observable case).

Remark 3.8. This is the only axiom that introduces a negative constraint: `zero` $\not\approx$ `one`, in particular `zero` $\neq$ `one`. Hence, a single point cannot be a model of LPT unless it chooses `Currency` empty. One receives easily that the expressions `one` $+ \ldots +$ `one`, where $k \in \mathbb{N}$ repetitions are meant, are pairwise different. Hence, any model of LPT is infinite.

3.1.4 `scale`

One expects `scale` to commute with any primitive. For the primitives considered so far, it suffices to require the following axiom to achieve this:

$$\texttt{scale}\ \alpha\ (x + y) \approx \texttt{scale}\ \alpha\ x + \texttt{scale}\ \alpha\ y \tag{*3.12}$$

`scale` should further represent multiplication:

$$\texttt{scale}\ \alpha\ (\texttt{scale}\ \beta\ x) \approx \texttt{scale}\ (\alpha \cdot \beta)\ x \tag{*3.13}$$

$$\texttt{scale}\ 0\ x \approx \texttt{zero} \tag{*3.14}$$

$$\texttt{scale}\ 1\ x \approx x \tag{*3.15}$$

And `scale` should further be monotonic:

$$x \preceq_b y \Rightarrow \texttt{scale}\ \alpha\ x \preceq_b \texttt{scale}\ \alpha\ y \tag{*3.16}$$

These axioms are justified intuitively by the fact that `scale` should just multiply all payments by a non-negative constant.

For `one`, note that `scale` α `one` just means "α dollars". Hence, one expects the simple fact that if a contract pays α dollars and β dollars, it pays a total of $\alpha + \beta$ dollars:

$$(\texttt{scale}\ \alpha\ \texttt{one}) + (\texttt{scale}\ \beta\ \texttt{one}) \approx \texttt{scale}\ (\alpha + \beta)\ \texttt{one} \tag{*3.17}$$

Remark 3.9. Note that this form of distributivity in the α argument does not hold in general:

$$(\texttt{scale}\ \alpha\ x) + (\texttt{scale}\ \beta\ x) \not\approx \texttt{scale}\ (\alpha + \beta)\ x$$

To see this, set $\alpha = \beta = 1$ and $x = y \vee (-y)$. – The LHS allows the contract $y - y \approx 0$ while the RHS only allows $\pm(2 \cdot y)$ which might both expose the holder to risk.[22] It is easy to construct an explicit counterexample in a probabilistic model like in section 5.

Hence, `scale` cannot be used to turn `Con` into a $\mathbb{R}$ vector space.

Lemma 3.10.

1. `scale` α `zero` $\approx$ `zero`
2. `scale` α (`give` x) $\approx$ `give` (`scale` α x)

Proof. For $\alpha = 0$, both statements are trivial via axiom (*3.14). For $\alpha > 0$, `scale` α is an invertible map commuting with the group operation `and`. Hence, it is already an automorphism and the statement follows. □

Remark 3.11. For $\alpha > 0$, now `scale` α is an automorphism, both of any group structure (`Con`, `zero`, `and`, `give`) up to "$\approx_b$" as well as any partial order "$\preceq_b$", with inverse `scale` $\frac{1}{\alpha}$.

By lemma 3.5 then `scale` α is also automorphism of the relations "$\prec_b$".

[22] The two sides *are* equal in present value if $y \succeq 0$ or $y \preceq 0$.

The laws above justify writing "$\cdot$" for `scale`. Keep in mind however that distributivity of sums on the $\mathbb{R}^+$ side is not given.

One receives that `one`, `scale` and "$\preceq$" work together in a sane way which is the first step towards compatibility of general prices with "$\preceq$":

Lemma 3.12. *Let $\alpha, \beta :: \mathbb{R}^+$. The following sets of statements are equivalent, respectively:*

1. $\alpha \cdot \texttt{one} \overset{\prec_b}{\underset{\preceq_b}{}} \beta \cdot \texttt{one}$ *for some* $b \neq \bot$.
2. $\alpha \cdot \texttt{one} \overset{\prec}{\underset{\preceq}{}} \beta \cdot \texttt{one}$.
3. $\alpha \overset{<}{\underset{\leq}{}} \beta$.

Proof. $(2 \Rightarrow 1)$ is trivial.

$(3 \Rightarrow 2)$: There is nothing to show for $\alpha = \beta$, so assume $\alpha < \beta$, i.e. $\beta - \alpha > 0$. As `scale` $(\beta - \alpha)$ preserves "$\prec$" and `one` $\succ 0$ by axiom (*3.11), $(\beta - \alpha) \cdot \texttt{one} \succ (\beta - \alpha) \cdot 0 \approx 0$. Now

$$\begin{aligned} \alpha \cdot \texttt{one} &\approx \alpha \cdot \texttt{one} + 0 \\ &\prec \alpha \cdot \texttt{one} + (\beta - \alpha) \cdot \texttt{one} \\ &\approx (\alpha + \beta - \alpha) \cdot \texttt{one} \\ &= \beta \cdot \texttt{one} \end{aligned}$$

where the second relation is because `and` $(\alpha \cdot \texttt{one})$ is an isomorphism and $0 \prec (\beta - \alpha) \cdot \texttt{one}$ and the third relation is due to axiom (*3.17).

$(1 \Rightarrow 3)$: If $\alpha \not\leq \beta$, i.e. $\beta < \alpha$, then by $(3 \Rightarrow 2)$, $\beta \cdot \texttt{one} \prec \alpha \cdot \texttt{one}$. In particular, by definition of "$\prec$", $\alpha \cdot \texttt{one} \not\preceq_b \beta \cdot \texttt{one}$.

If $\alpha \not< \beta$, i.e. $\beta \leq \alpha$, then again by $(3 \Rightarrow 2)$, $\beta \cdot \texttt{one} \preceq \alpha \cdot \texttt{one}$. In particular, $\beta \cdot \texttt{one} \preceq_b \alpha \cdot \texttt{one}$, so $\alpha \cdot \texttt{one} \not\prec_b \beta \cdot \texttt{one}$. □

Corollary 3.13. *Lemma 3.12 still holds if one allows $\alpha, \beta :: \mathbb{R}$ instead of only $\mathbb{R}^+$.*

Proof. I only show the "$\preceq$" variant and only $(2 \Leftrightarrow 3)$. The other parts are similar.

For general α and β, "$\cdot$" means `scaleG` instead of `scale`. There are four cases:

1. $\alpha, \beta \geq 0$. Then the statement follows by lemma 3.12.
2. $\alpha, \beta < 0$.

$$\begin{aligned} & \texttt{scaleG}\ \alpha\ \texttt{one} \preceq \texttt{scaleG}\ \beta\ \texttt{one} \\ \Leftrightarrow\ & -((-\alpha) \cdot \texttt{one}) \preceq -((-\beta) \cdot \texttt{one}) \\ \Leftrightarrow\ & (-\alpha) \cdot \texttt{one} \succeq (-\beta) \cdot \texttt{one} \\ \Leftrightarrow\ & -\alpha \geq -\beta \\ \Leftrightarrow\ & \alpha \leq \beta \end{aligned}$$

   where the third equivalence is by lemma 3.12.

3. $\alpha < 0 \leq \beta$. Then obviously $\alpha < \beta$ and

$$\texttt{scaleG}\ \alpha\ \texttt{one} = -((-\alpha) \cdot \texttt{one}) \prec 0 \preceq \beta \cdot \texttt{one} = \texttt{scaleG}\ \beta\ \texttt{one}.$$

4. $\beta < 0 \leq \alpha$. Just like the previous case. □

3.1.5 `or`

`or` $x\ y = x \vee y$ should be the *join* of x and y with respect to all the partial orders "$\preceq_b$", i.e.

$$x \preceq x \vee y \text{ and } y \preceq x \vee y \tag{*3.18}$$

$$x \preceq_b z \text{ and } y \preceq_b z \Rightarrow x \vee y \preceq_b z. \tag{*3.19}$$

This can be justified as follows: (*3.18) follows from the fact that $x \vee y$ can model both x and y by making the according choice. For (*3.19), assume that z is as in the axiom and in some scenario b holds and the price of $x \vee y$ is strictly greater than the price of z. Then an arbitrageur would sell $x \vee y$ and buy z, thus making the price difference as a profit. The counterparty can choose between x and y, and both cases can be made risk-free without additional cost by assumption.

Two subtle assumptions are made here: An arbitrageur can rely on the fact that the counterparty chooses "first" if z contains choice as well and no time is required to communicate this choice.

As usual, joins are unique (up to "$\approx$"), so there is only one possible value for $x \vee y$ up to present value.

Lemma 3.14. *Let* $f :: \texttt{Con} \to \texttt{Con}$ *be an automorphism of a preorder "*$\preceq_b$*". Then*

$$f\ (x \vee y) \approx_b (f\ x) \vee (f\ y).$$

If $x :: \texttt{Con}$*, then* (`or` x) *is a homomorphism of any preorder "*$\preceq_b$*".*

Proof. These standard theorems follow directly from the universal property of the join. □

Lemma 3.15.

1. $x + (y \vee z) \approx (x + y) \vee (x + z)$
2. $\alpha \cdot (x \vee y) \approx \alpha \cdot x \vee \alpha \cdot y$.

Proof. Both statements follow from lemma 3.14 for $b = \top$:

As seen above, both `and` x and `scale` α for $\alpha > 0$ are automorphisms of "$\preceq$". For $\alpha = 0$, the second statement is trivial. □

Remark 3.16. In lemma 3.15.1, the symbols "+" and "$\vee$" cannot be interchanged. In general, there is no simple relation between

$$x \vee (y + z) \text{ vs. } (x \vee y) + (x \vee z).$$

To see this, consider the following multiples of `one` for (x, y, z):

- $(1, -1, -1)$. Then the LHS reduces to $1 \vee -2 \approx 1$ and the RHS reduces to $1 + 1 \approx 2$, hence the RHS is strictly greater (via lemma 3.12).
- $(-1, -1, -1)$. Then the LHS reduces to $-1 \vee -2 \approx -1$ and the RHS reduces to $-1 + -1 \approx -2$, hence the LHS is strictly greater.

One receives an equation similar to axiom (*3.17) directly from the universal property of the join:

Lemma 3.17.

$$(\texttt{scale}\ \alpha\ \texttt{one}) \vee (\texttt{scale}\ \beta\ \texttt{one}) \approx \texttt{scale}\ (\max(\alpha, \beta))\ \texttt{one}$$

Here, $\max(\alpha, \beta)$ is short for $\texttt{if}'\ (\alpha \leq \beta)\ \alpha\ \beta$, of course.

Proof. Assume wlog. that $\alpha \leq \beta$ (otherwise, swap α and β). Then $\texttt{scale}\ \alpha\ \texttt{one} \preceq \texttt{scale}\ \beta\ \texttt{one}$ by lemma 3.12 and hence the LHS is in present value equal to $\texttt{scale}\ \beta\ \texttt{one}$, which is the RHS. □

The dual notion to the join $x \vee y$ is the *meet* $x \wedge y$, i.e. the greatest common lower bound. The following lemma shows that it is attained by the contract where the counterparty chooses which of x or y the holder of $x \wedge y$ should acquire.

Lemma 3.18. *For $x, y :: \texttt{Con}$, let $x \wedge y = -(-x \vee -y)$. Then $x \wedge y$ is the* meet *of x and y in any partial order "$\preceq_b$", i.e. the following hold:*

1. *$x \succeq x \wedge y$ and $y \succeq x \wedge y$*
2. *$x \succeq_b z$ and $y \succeq_b z \Rightarrow x \wedge y \succeq_b z$ for any $b :: \texttt{OB}$.*

Proof. The proof is similar to the one of lemma 3.14, just `give` is a bijective map that *flips* the ordering instead of preserving it:

$-x \preceq -x \vee -y$, hence $x \approx - - x \succeq -(-x \vee -y) = x \wedge y$. Analogous for y.

Let $z \preceq x, y$. Then $-z \succeq -x, -y$, hence $-z \succeq -x \vee -y$, hence $z \preceq -(-x \vee -y) = x \wedge y$. □

3.1.6 `when′`

The following two combinators `when′` and `anytime` will introduce time delays on contracts. These are the more interesting cases which would result in stochastic integrals and the such when *computing* present values.[23] The axioms introduced need to be more sophisticated as well to account for "history": If $x \preceq_b y$, then one does *not* receive $\texttt{when}'\ c\ x \preceq_b \texttt{when}'\ c\ y$. One *does* receive this in case $x \preceq_c y$, but that would be too weak. Instead, the monotonicity rules for `when′` are defined in terms of $\mathfrak{e}$ and $\mathfrak{a}$.

First of all, the above primitives which do not introduce choice should commute with `when′`:

$$\texttt{when}'\ b\ 0 \approx 0 \tag{*3.20}$$

$$\texttt{when}'\ b\ (x + y) \approx \texttt{when}'\ b\ x + \texttt{when}'\ b\ y \tag{*3.21}$$

$$\texttt{when}'\ b\ (\alpha \cdot x) \approx \alpha \cdot (\texttt{when}'\ b\ x) \tag{*3.22}$$

These are justified easily. For (*3.21) one needs to take into consideration that $\texttt{when}'\ b$ itself does not introduce choice.

Recap that I also write $\mathfrak{W}'$ for `when′` and that $\mathfrak{W}'\ b\ x$ acquires x immediately if b has *ever before* been true. The following axioms define the actual behavior of `when′`:

$$\mathfrak{W}'\ b\ x \approx_{\mathfrak{e}\ b} x \tag{*3.23}$$

$$x \underset{\mathfrak{e}\ d \wedge b}{\preceq} \mathfrak{W}'\ c\ y \text{ and } \mathfrak{W}'\ b\ x \underset{\mathfrak{e}\ d \wedge c}{\preceq} y \Rightarrow \mathfrak{W}'\ b\ x \underset{\mathfrak{e}\ d \wedge \neg \mathfrak{e}\ b \wedge \neg \mathfrak{e}\ c}{\preceq} \mathfrak{W}'\ c\ y \tag{*3.24}$$

[23] Cf. section 5.2 for a simple case.

The first one is clear: In situations where $\mathfrak{e}\ b$ is true, x is immediately acquired, hence $\mathfrak{W}'\ b\ x$ is the same as x.

To see that the second one must hold, assume that the premise is true and consider a situation where $\mathfrak{e}\ d$ holds and b and c have both never been true (such that none of $\mathfrak{W}'\ b\ x$ or $\mathfrak{W}'\ c\ y$ is triggered immediately). Assume that in some scenario, the price, i.e. the total cost of acquisition, of $\mathfrak{W}'\ b\ x$ is strictly higher than that of $\mathfrak{W}'\ c\ y$. Consider a trader buying $\mathfrak{W}'\ c\ y$ and selling $\mathfrak{W}'\ b\ x$, thus making the price difference as a profit.

The resulting position can be made risk-free, thus result in arbitrage, by the following strategy:

1. Do nothing until the first of b or c becomes true. This might be never. Wlog. assume that b becomes true first.
2. The resulting position is now equivalent to holding $-x$ and $\mathfrak{W}'\ c\ y$: y hasn't been acquired before as $\mathfrak{e}\ c$ hadn't been true and x has been acquired just at this moment.[24] Also, $\mathfrak{e}\ d$ is still true because it was true at acquisition time already and b is true by assumption.
3. By assumption, $x \preceq_{\mathfrak{e}\ d\wedge b} \mathfrak{W}'\ c\ y$, so by buying x and selling $\mathfrak{W}'\ c\ y$, one can arrive at a risk-free position without cost.

Remark 3.19. Via case distinction (axiom (*3.4)), one receives the following variants of axiom (*3.24):

$$\underset{\mathfrak{e}\ d\wedge\mathfrak{e}\ b}{x \preceq} \mathfrak{W}'\ c\ y \text{ and } \mathfrak{W}'\ b\ \underset{\mathfrak{e}\ d\wedge\mathfrak{e}\ c}{x \preceq} y \Leftrightarrow \mathfrak{W}'\ b\ \underset{\mathfrak{e}\ d}{x \preceq} \mathfrak{W}'\ c\ y \tag{3.25}$$

$$\underset{\mathfrak{e}\ d\wedge b}{x \preceq} \mathfrak{W}'\ c\ y \text{ and } \mathfrak{W}'\ b\ \underset{\mathfrak{e}\ d\wedge c}{x \preceq} y \Rightarrow \mathfrak{W}'\ b\ \underset{\mathfrak{e}\ d\wedge\neg\bar{\mathfrak{e}}\ b\wedge\neg\bar{\mathfrak{e}}\ c}{x \preceq} \mathfrak{W}'\ c\ y \tag{3.26}$$

$$\underset{\mathfrak{e}\ d\wedge b}{x \preceq} \mathfrak{W}'\ c\ y \text{ and } \mathfrak{W}'\ b\ \underset{\mathfrak{e}\ d\wedge c}{x \preceq} y \Rightarrow \mathfrak{W}'\ b\ \underset{\mathfrak{e}\ d\wedge(\neg\mathfrak{e}\ b\vee b)\wedge(\neg\mathfrak{e}\ c\vee c)}{x \preceq} \mathfrak{W}'\ c\ y \tag{3.27}$$

$$\underset{d\wedge b}{x \preceq} \mathfrak{W}'\ c\ y \text{ and } \mathfrak{W}'\ b\ \underset{d\wedge c}{x \preceq} y \Leftrightarrow \mathfrak{W}'\ b\ \underset{d\wedge(b\vee c)}{x \preceq} \mathfrak{W}'\ c\ y \tag{3.28}$$

For the second one, use $\neg\mathfrak{e}\ b \Rightarrow \neg\bar{\mathfrak{e}}\ b \Rightarrow \neg\mathfrak{e}\ b \vee b$. The last statement is trivial.

The converse of (*3.24) is not true: Consider $b = c = d = \top$. Then $\neg\mathfrak{e}\ b = \neg\mathfrak{e}\ c = \bot$ and so the RHS is trivially true, but the LHS is equivalent to $x \preceq y$. The same argument works for (3.26) and (3.27).

Lemma 3.20.

1. $\mathfrak{W}'\ b\ (-x) \approx -\mathfrak{W}'\ b\ x$
2. $\mathfrak{W}'\ \bot\ x \approx 0$
3. *If* $\mathfrak{e}\ d \Rightarrow (\mathfrak{e}\ b \leftrightarrow \mathfrak{e}\ c)$, *then* $\mathfrak{W}'\ b\ x \approx_{\mathfrak{e}\ d} \mathfrak{W}'\ c\ x$.
   If $\mathfrak{e}\ d \Rightarrow \neg\mathfrak{e}\ b$, *then* $\mathfrak{W}'\ b\ x \approx_{\mathfrak{e}\ d} 0$.
4. $\mathfrak{W}'\ b\ x \approx \mathfrak{W}'\ (\mathfrak{e}\ b)\ x$

Proof. 1: By the axioms (*3.20) and (*3.21), $\mathfrak{W}'\ b$ is a group homomorphism of $(\texttt{Con}, \texttt{zero}, \texttt{and}, \texttt{give})$. Then the statement follows from the uniqueness of the inverse (up to present value).

[24] The special case where both $\mathfrak{e}\ b$ and $\mathfrak{e}\ c$ become true at the exact same time is covered here: Then $\mathfrak{W}'\ c\ y$ is y.

2: Apply (3.25) to $b = c = \bot$, $d = \top$, and $y = 0$. We have $x \preceq_\bot \mathfrak{W}' \bot 0$ and $\mathfrak{W}' \bot x \preceq_\bot 0$ because "$\preceq_\bot$" is trivial. Hence, $\mathfrak{W}' \bot x \preceq \mathfrak{W}' \bot 0$ and $\mathfrak{W}' \bot 0 \approx 0$ by axiom (*3.20). The "$\succeq$" direction is analogous.

3: For the first statement, note how the premise implies that $\mathfrak{e}\ d \wedge \mathfrak{e}\ b \Rightarrow \mathfrak{e}\ c$ and $x \approx_{\mathfrak{e}\ c} \mathfrak{W}'\ c\ x$ by axiom (*3.23), hence $x \approx_{\mathfrak{e}\ d \wedge \mathfrak{e}\ b} \mathfrak{W}'\ c\ x$. Analogously, one receives $\mathfrak{W}'\ b\ x \approx_{\mathfrak{e}\ d \wedge \mathfrak{e}\ c} x$. The conclusion then follows by (3.25).

For the second statement of part 3, apply the first one to $c = \bot$ and use part 2.

4: Apply part 3 to $d = \top$ and $c = \mathfrak{e}\ b$: We have $\mathfrak{e}\ c = \mathfrak{e}\ (\mathfrak{e}\ b) = \mathfrak{e}\ b$. □

Note how in part 3 of the previous lemma, we do *not* in general receive equality in present value under conditions $\mathfrak{e}\ b \leftrightarrow \mathfrak{e}\ c$ (consider $\neg\mathfrak{e}\ b \wedge \neg\mathfrak{e}\ c$), but only for conditions of "$\mathfrak{e}$" form.

Lemma 3.21 ($\mathfrak{W}'$ monotonicity)**.**

1. *If* $x \preceq_{\mathfrak{e}\ d \wedge \mathfrak{e}\ b} y$, *then* $\mathfrak{W}'\ b\ x \preceq_{\mathfrak{e}\ d} \mathfrak{W}'\ b\ y$.
2. *If* $x \preceq_{\mathfrak{e}\ d \wedge b} y$, *then* $\mathfrak{W}'\ b\ x \preceq_{\mathfrak{e}\ d \wedge \neg\bar{\mathfrak{e}}\ b} \mathfrak{W}'\ b\ y$.

Proof. 1: We have $x \preceq_{\mathfrak{e}\ d \wedge \mathfrak{e}\ b} y \approx_{\mathfrak{e}\ b} \mathfrak{W}'\ b\ y$. Analogously, $\mathfrak{W}'\ b\ x \preceq_{\mathfrak{e}\ d \wedge \mathfrak{e}\ b} y$. Then by (3.25), the conclusion follows.

2: Apply the same consideration to (3.26) with b in place of $\mathfrak{e}\ b$. □

Lemma 3.22. $\mathfrak{W}'\ b\ (\mathfrak{W}'\ c\ x) \approx \mathfrak{W}'\ (\mathfrak{e}\ b \wedge \mathfrak{e}\ c)\ x$.

Proof. Let $f = \mathfrak{e}\ b \wedge \mathfrak{e}\ c$. It is easy to see that $f = \mathfrak{e}\ f$. By (3.25), one needs to show the following:

1. $\mathfrak{W}'\ b\ (\mathfrak{W}'\ c\ x) \approx_f x$
2. $\mathfrak{W}'\ c\ x \approx_{\mathfrak{e}\ b} \mathfrak{W}'\ f\ x$

For the first statement, as f implies both $\mathfrak{e}\ b$ and $\mathfrak{e}\ c$, we have by axiom (*3.23):

$$\mathfrak{W}'\ b\ (\mathfrak{W}'\ c\ x) \approx_f \mathfrak{W}'\ c\ x \approx_f x.$$

For the second statement, via lemma 3.20.3, one notes that

$$\mathfrak{e}\ b \Rightarrow (\mathfrak{e}\ c \leftrightarrow \mathfrak{e}\ f)$$

because $(\mathfrak{e}\ c \leftrightarrow \mathfrak{e}\ f) = (\mathfrak{e}\ c \leftrightarrow f) = (\mathfrak{e}\ c \leftrightarrow (\mathfrak{e}\ b \wedge \mathfrak{e}\ c))$. □

Remark 3.23. `when′ b` does *not* commute with `or`: For example, let o :: `Obs` $\mathbb{R}$ be some observable and b :: `OB` and consider

$$\mathfrak{W}'\ b\ (o \vee 0) \text{ vs. } \mathfrak{W}'\ b\ o \vee \mathfrak{W}'\ b\ 0 \approx \mathfrak{W}'\ b\ 0 \vee 0.$$

On the RHS, the holder must make a choice at acquisition time. If she chooses $\mathfrak{W}'\ b\ o$, the value of o might be negative at time b, exposing her to risk. On the LHS, she could in this case simply choose 0. An explicit counterexample is easily constructed in a model such as in section 5. – Unless the fully deterministic special case is considered, which is a LPT model, so LPT does not prove the existence of a counterexample.

One direction of the above comparison always holds, namely that choosing later, i.e. with more information available, is always better:

Lemma 3.24.

$$\mathfrak{W}'\ b\ (x \vee y) \succeq \mathfrak{W}'\ b\ x \vee \mathfrak{W}'\ b\ y$$

Proof. $x \vee y \succeq x$, so by monotonicity $\mathfrak{W}'\ b\ (x \vee y) \succeq \mathfrak{W}'\ b\ x$. Likewise for y. Then the claim follows by the universal property of the join. □

3.1.7 `anytime`

Imagine a market participant acquiring $\mathfrak{A}\ x$: She can either exercise the option, thus acquiring x, decide never to exercise the option, which would be equivalent to exchanging it for the `zero` contract, or postpone the decision. Postponing means waiting for a certain event $b :: \texttt{OB}$ to occur,[25] i.e. exchanging $\mathfrak{A}\ x$ for $\mathfrak{W}'\ b\ (\mathfrak{A}\ x)$.

Following this discussion, $\mathfrak{A}\ x$ should be valued higher than x and than $\mathfrak{W}'\ b\ (\mathfrak{A}\ x)$ for any b because it can reduce to these contracts and be minimal with this property because it cannot do anything else:

$$\mathfrak{A}\ x \succeq x \tag{*3.29}$$

$$\forall b :: \texttt{OB} :\ \mathfrak{A}\ x \succeq \mathfrak{W}'\ b\ (\mathfrak{A}\ x) \tag{*3.30}$$

$$(z \succeq_{\mathfrak{e}\ d} x \text{ and } \forall b :: \texttt{OB} :\ z \succeq_{\mathfrak{e}\ d} \mathfrak{W}'\ b\ z) \Rightarrow z \succeq_{\mathfrak{e}\ d} \mathfrak{A}\ x \tag{*3.31}$$

The argument for minimality is weaker here than for the other primitives: In order to construct an portfolio, an arbitrageur would have to know the strategy the counterparty is following, i.e. the event b they wait for. However, the results, especially Merton's theorem 4.34, suggest that the axiom is chosen correctly.

As for `when′`, $\mathfrak{e}\ d$ here acts as a side condition which stays true if it was true at acquisition.

Remark 3.25.

1. Uniqueness of $\mathfrak{A}\ x$ up to present value follows again because it is given by a universal property.
2. By setting $b = \bot$ in (*3.30), we also receive

$$\mathfrak{A}\ x \succeq 0. \tag{3.32}$$

One first of all receives the expected monotonicity result similar to `when′`:

Lemma 3.26. *If* $x \preceq_{\mathfrak{e}\ d} y$, *then* $\mathfrak{A}\ x \preceq_{\mathfrak{e}\ d} \mathfrak{A}\ y$.

Proof. Use (*3.31) with respect to $\mathfrak{A}\ x$ and $z = \mathfrak{A}\ y$:

- $\mathfrak{A}\ y \succeq y \succeq_{\mathfrak{e}\ d} x$ by assumption.

[25] Of course, it is a design decision to model it like this. Note that in discrete time, there is really only one relevant b per time step, namely waiting for the next point in time, a statement which will be made precise in section 3.4. In continuous time, it might be argued that a human trader could choose to exercise "arbitrarily" instead of waiting for a certain event, which could mean e.g. that the holder waits for an event which is however not observable, like an internal condition of her private company.

- $\mathfrak{W}'\ b\ (\mathfrak{A}\ y) \preceq \mathfrak{A}\ y$ by (*3.30). □

For the commutativity results, I first show a technical lemma:

Lemma 3.27. *Let $f :: \texttt{Con} \to \texttt{Con}$ be a homomorphism of some partial order "$\preceq_{\mathfrak{e}\ d}$" such that $f\ (\mathfrak{W}'\ b\ x) \approx_{\mathfrak{e}\ d} \mathfrak{W}'\ b\ (f\ x)$ for any $x :: \texttt{Con}$ and $b :: \texttt{OB}$.*
Then $\mathfrak{A}\ (f\ x) \preceq_{\mathfrak{e}\ d} f\ (\mathfrak{A}\ x)$.
If f even an automorphism up to "$\approx_{\mathfrak{e}\ d}$", then $\mathfrak{A}\ (f\ x) \approx_{\mathfrak{e}\ d} f\ (\mathfrak{A}\ x)$.

Proof. For the first part, it suffices to show that $f\ (\mathfrak{A}\ x)$ has the properties for z from axiom (*3.31):

1. $\mathfrak{A}\ x \succeq x$, so by monotonicity $f\ (\mathfrak{A}\ x) \succeq_{\mathfrak{e}\ d} f\ x$.
2. For $b :: \texttt{OB}$ we have by assumption $\mathfrak{W}'\ b\ (f\ (\mathfrak{A}\ x)) \preceq_{\mathfrak{e}\ d} f\ (\mathfrak{W}'\ b\ (\mathfrak{A}\ x)) \preceq_{\mathfrak{e}\ d} f\ (\mathfrak{A}\ x)$.

For the second part, note that f^{-1} fulfills the assumption for this lemma as well and hence we have

$$\mathfrak{A}\ x \approx_{\mathfrak{e}\ d} \mathfrak{A}\ \left(f^{-1}\ (f\ x)\right) \preceq_{\mathfrak{e}\ d} f^{-1}\ (\mathfrak{A}\ (f\ x)).$$

By applying f on both sides, the claim follows. □

Now it is easily seen that `anytime` commutes with `zero` and `scale` α:

Lemma 3.28.

1. $\mathfrak{A}\ 0 \approx 0$.

2. $\mathfrak{A}\ (\alpha \cdot x) \approx \alpha \cdot (\mathfrak{A}\ x)$.

Proof. 1: $\mathfrak{A}\ 0 \succeq 0$ by (3.32). For "$\preceq$", apply lemma 3.27 to the homomorphism $\lambda x.\ 0$ and $d = \top$.
2: For $\alpha = 0$, the statement is trivial. For $\alpha > 0$, apply lemma 3.27 to the automorphism $\lambda x.\ \alpha \cdot x$. □

Remark 3.29. One might expect `anytime` to commute with `and`, but that is not the case. Comparing the contracts $\mathfrak{A}\ (x + y)$ and $\mathfrak{A}\ x + \mathfrak{A}\ y$, a holder of the latter can choose when to acquire x and y *independently*, while for the former, they must be acquired at the same time.

A counterexample can be constructed using variants of the `read′` primitive as follows. The functions used can be found in figure 7. Their semantics require axioms from the following section 3.1.8.

Let $t_1 \neq t_2 :: \texttt{Time}$, $x_i = \texttt{cond}\ (\texttt{now} = t_i)\ \texttt{one}\ \texttt{zero}$ for $i = 1, 2$. Let $d = (\texttt{now} = \min(t_1, t_2))$. Then

$$\begin{aligned} \mathfrak{A}\ x_1 + \mathfrak{A}\ x_2 \succeq\ & \texttt{at}\ t_1\ x_1 + \texttt{at}\ t_2\ x_2 \\ \approx_d\ & \texttt{one} + \texttt{at}\ t_2\ \texttt{one} \end{aligned}$$

On the other hand, since $x_1 + x_2 \preceq \texttt{one}$ by $t_1 \neq t_2$ we have $\mathfrak{A}\ (x_1 + x_2) \preceq \mathfrak{A}\ \texttt{one}$.

If interest rates exist and are non-negative (definition 4.4), then $\mathfrak{A}\ \texttt{one} \approx \texttt{one}$ and $\texttt{one} + \texttt{at}\ t_2\ \texttt{one} \succ \texttt{one}$, so the two contracts are not equal in present value.

One receives that making choices separately is always better and that making choices at exertion is always better than at acquisition:

Lemma 3.30.

$$\mathfrak{A}\ (x + y) \preceq \mathfrak{A}\ x + \mathfrak{A}\ y$$
$$\mathfrak{A}\ (x \vee y) \succeq \mathfrak{A}\ x \vee \mathfrak{A}\ y$$

Proof. First part: I show that $\mathfrak{A}\ x + \mathfrak{A}\ y$ satisfies the preconditions of axiom (*3.31) for $\mathfrak{A}\ (x + y)$:

- $\mathfrak{A}\ x \succeq x$ and $\mathfrak{A}\ y \succeq y$, so $\mathfrak{A}\ x + \mathfrak{A}\ y \succeq x + y$.
- $\mathfrak{W}'\ b\ (\mathfrak{A}\ x + \mathfrak{A}\ y) \approx \mathfrak{W}'\ b\ (\mathfrak{A}\ x) + \mathfrak{W}'\ b\ (\mathfrak{A}\ y) \preceq \mathfrak{A}\ x + \mathfrak{A}\ y$.

Second part: Since $x \vee y \succeq x$, one has by monotonicity also $\mathfrak{A}\ (x \vee y) \succeq \mathfrak{A}\ x$. Analogous for y. Hence, by the universal property of "$\vee$", $\mathfrak{A}\ (x \vee y) \succeq \mathfrak{A}\ x \vee \mathfrak{A}\ y$. □

`anytime` commutes with `when'` c:

Lemma 3.31.

$$\mathfrak{A}\ (\mathfrak{W}'\ c\ x) \approx \mathfrak{W}'\ c\ (\mathfrak{A}\ x)$$

The intuition behind this statement is that if the `anytime` option at the LHS is exercised early, one would have to wait for $\mathfrak{e}\ c$ anyway, so there is no point in doing that, and if it is exercised when $\mathfrak{e}\ c$ is true, then x is acquired immediately. So the LHS should be equivalent to first waiting for $\mathfrak{e}\ c$, then receiving the option, which is what the RHS does.

Proof. "$\preceq$": Apply lemma 3.27 to $\lambda x.\ \mathfrak{W}'\ c\ x$. This is monotonic with respect to "$\preceq$" and for any $b :: \texttt{OB}$ we have by lemma 3.22

$$\mathfrak{W}'\ b\ (\mathfrak{W}'\ c\ x) \approx \mathfrak{W}'\ (\mathfrak{e}\ b \wedge \mathfrak{e}\ d)\ x \approx \mathfrak{W}'\ c\ (\mathfrak{W}'\ b\ x),$$

so the preconditions are fulfilled.

"$\succeq$": By axiom (*3.23) and monotonicity of $\mathfrak{W}'\ c$ and $\mathfrak{A}$ (lemmas 3.21 and 3.26) we receive

$$\begin{aligned} & \mathfrak{W}'\ c\ x \approx_{\mathfrak{e}\ c} x \\ \Rightarrow \quad & \mathfrak{A}\ (\mathfrak{W}'\ c\ x) \approx_{\mathfrak{e}\ c} \mathfrak{A}\ x \\ \Rightarrow \quad & \mathfrak{W}'\ c\ (\mathfrak{A}\ (\mathfrak{W}'\ c\ x)) \approx \quad \mathfrak{W}'\ c\ (\mathfrak{A}\ x) \end{aligned}$$

and $\mathfrak{W}'\ c\ (\mathfrak{A}\ (\mathfrak{W}'\ c\ x)) \preceq \mathfrak{A}\ (\mathfrak{W}'\ c\ x)$ by axiom (*3.30). □

Finally, $\mathfrak{A}$ is idempotent. This is intuitively clear: An option which does nothing but acquire another option can be collapsed.

Lemma 3.32.

$$\mathfrak{A}\ (\mathfrak{A}\ x) \approx \mathfrak{A}\ x$$

Proof. "$\succeq$" is axiom (*3.29). For "$\preceq$", apply minimality (axiom (*3.31)): We have $\mathfrak{A}\ x \succeq \mathfrak{A}\ x$ trivially and $\mathfrak{A}\ x \succeq \mathfrak{W}'\ b\ (\mathfrak{A}\ x)$ for all b by axiom (*3.30). □

Remark 3.33. Define $\mathfrak{E}\ x := -\mathfrak{A}\ (-x)$. $\mathfrak{E}$ could also be written "`sometime`" in contrast to `anytime`: The counterparty decides when the holder will acquire x.

It is easy to see that as in lemma 3.18, $\mathfrak{E}$ will have all the properties of $\mathfrak{A}$ with "$\preceq$" and "$\succeq$" interchanged. I.e. the following holds:

$$\mathfrak{E}\ x \preceq x \tag{3.33}$$

$$\forall b :: \mathtt{OB} :\ \mathfrak{E}\ x \preceq \mathfrak{W}'\ b\ (\mathfrak{E}\ x) \tag{3.34}$$

$$(z \preceq_{\mathfrak{c}\ d} x \text{ and } \forall b :: \mathtt{OB} :\ z \preceq_{\mathfrak{c}\ d} \mathfrak{W}'\ b\ z) \Rightarrow z \preceq_{\mathfrak{c}\ d} \mathfrak{E}\ x \tag{3.35}$$

For the intuition here, x should be thought of as being negative, i.e. the holder of $\mathfrak{E}\ x$ would want to avoid acquiring x. One can show by methods above that if $x \succeq 0$, then $\mathfrak{E}\ x \approx 0$.

3.1.8 `read′`

`read′` presents an interface between contracts and observables. To find axioms for `read′`, one first notes how `read′` is similar to `join`:

`join` :: `Obs` (`Obs` a) $\to$ `Obs` a : `join` o defines an observable that, when read, reads o at that same time and then reads the resulting observable.

`read′` :: `Obs Con` $\to$ `Con` : `read′` p defines a contract that, when acquired, reads p at that same time and then acquires the resulting contract.

One now expects laws similar to those of `join` to hold for `read′` as well. The axioms (*Mo3) and (*Mo4) can be made well-typed with `read′` in place of `join` as follows:

$$\mathtt{read'} \circ \mathtt{join} \approx \mathtt{read'} \circ \mathtt{fmap\ read'} \tag{*3.36}$$

$$\mathtt{read'} \circ \mathtt{return} \approx \mathtt{id} \tag{*3.37}$$

The following axiom guarantees compatibility with `fmap`: For $i = 1, 2$ let a_i be a type such that equality on a_i is lifted to functions.[26] Let $f_i :: a_i \to \mathtt{Con}$, $o_i :: \mathtt{Obs}\ a_i$ and $d :: \mathtt{OB}$. Then the following should hold:

$$\left(\forall x_1 :: a_1, x_2 :: a_2 :\ f_1\ x_1 \preceq f_2\ x_2 \underset{d \wedge o_1 = x_1 \wedge o_2 = x_2}{}\right) \Rightarrow o_1 \rightsquigarrow f_1\ \preceq_d\ o_2 \rightsquigarrow f_2 \tag{*3.38}$$

Recap that $o \rightsquigarrow f = \mathtt{read'}\ (\mathtt{fmap}\ f\ o)$, so this is indeed a rule for `read′` and `fmap`.

The axioms should be intuitively clear from the intended meaning of `read′` and "$\rightsquigarrow$". The condition $(o_1 = x_1) \wedge (o_2 = x_2)$ in (*3.38) is used to transport dependencies between the observables o_1 and o_2. If, to use the simplest case as an example, $o_1 = o_2$, then the above condition is $(o_1 = x_1) \wedge (o_1 = x_2)$, which is $\bot$ – and hence the premise is trivially true – unless $x_1 = x_2$. One can see this by a standard lift reduction technique as in section 2.1. Hence, it suffices to consider the case $x_1 = x_2$.

The following lemmas show that the above axioms indeed make `read′` and `join` compatible in an intuitive sense. Many of the following lemmas have a counterpart in observables where "$\gg\!=$" has been replaced by "$\rightsquigarrow$" and "$b \Rightarrow \ldots$" has been replaced by "$\approx_b$".

[26] For details of functional lifts of relations cf. appendix A.1.4. In short, a_i must not be of form `Obs` a or `Con`.

Lemma 3.34. *Let* $f :: a \to \texttt{Con}$, $g :: a \to \texttt{Obs Con}$ *and* $h :: b \to \texttt{Obs}\ a$. *Let* $o :: \texttt{Obs}\ a$ *and* $p :: \texttt{Obs}\ b$. *Then*

1. $(\texttt{return}\ x) \rightsquigarrow f \approx f\ x$ *for all* $x :: a$
2. $\texttt{read}'\ (o \gg\!= g) \approx o \rightsquigarrow (\texttt{read}' \circ g)$
3. $(p \gg\!= h) \rightsquigarrow f \approx p \rightsquigarrow (\lambda x.\ (h\ x) \rightsquigarrow f)$

Proof. 1: By definition of "$\rightsquigarrow$", the LHS is equal to $\texttt{read}'\ (\texttt{fmap}\ f\ (\texttt{return}\ x))$ $= \texttt{read}'\ (\texttt{return}\ (f\ x))$ by axiom (*Mo1) for observables, which is equal in present value to $f\ x$ by axiom (*3.37).

2:

$$\begin{aligned}\lambda o.\ o \rightsquigarrow (\texttt{read}' \circ g) &= \texttt{read}' \circ \texttt{fmap}\ (\texttt{read}'\ \circ g)\\ &= \texttt{read}' \circ \texttt{fmap}\ \texttt{read}' \circ \texttt{fmap}\ g\\ &\approx \texttt{read}' \circ \texttt{join} \circ \texttt{fmap}\ g\\ &= \lambda o.\ \texttt{read}'\ (o \gg\!= g)\end{aligned}$$

where the equalities all follow by definition or from the monad and functor laws and the middle equality in present value holds by axiom (*3.36).

3:

$$\begin{aligned}(p \gg\!= h) \rightsquigarrow f &= \texttt{read}'\ (\texttt{fmap}\ f\ (p \gg\!= h))\\ &= \texttt{read}'\ (p \gg\!= (\texttt{fmap}\ f \circ h))\\ &\approx p \rightsquigarrow (\texttt{read}' \circ \texttt{fmap}\ f \circ h)\\ &= p \rightsquigarrow (\lambda x.\ (h\ x) \rightsquigarrow f)\end{aligned}$$

where the middle equality in present value follows from 2. □

Lemma 3.35. *Let* $f :: a \to \texttt{Con}$ *and* $o :: \texttt{Obs}\ a$ *such that equality is lifted for the type* a.

1. *If* $\forall x :: a : f\ x \preceq_{d \wedge o=x} g\ x$, *then* $o \rightsquigarrow f \preceq_d o \rightsquigarrow g$.
2. $o \rightsquigarrow f \approx_{o=x} f\ x$ *for all* $x :: a$.
3. *If* $z :: \texttt{Con}$ *and for all* $x :: a$ *we have* $f\ x \preceq_{d \wedge o=x} z$, *then* $o \rightsquigarrow f \preceq_d z$. *The analogous statement holds for "*$\succeq$*".*
4. *If for some* $x :: a$ *we have* $f \approx (\texttt{const}\ x)$, *then* $o \rightsquigarrow f \approx x$.

Proof. 1: By axiom (*3.38), one needs to show:

$$\forall x_1, x_2 :: a : \underset{d \wedge o=x_1 \wedge o=x_2}{f\ x_1 \preceq g\ x_2}$$

If $x_1 \neq x_2$, it is seen by lift reduction that

$$\begin{aligned}(o = x_1 \wedge o = x_2) &= \texttt{fmap}\ (\lambda x.\ x = x_1 \wedge x = x_2)\ o\\ &= \texttt{fmap}\ (\texttt{const}\ \texttt{False})\ o = \bot,\end{aligned}$$

so in this case, the above condition is trivially true. If $x_1 = x_2$, the condition true by assumption.

2: By 3.34.1, the RHS is equal (in present value) to $(\texttt{return}\ x) \rightsquigarrow f$. So by axiom (*3.38), it suffices to show that

$$\forall y, z :: a : f\ y \approx_{b(y,z)} f\ z$$

where $b(y,z) := (o = x \wedge o = y \wedge (\texttt{return}\ x) = z)$. If $y = z$, the above statement is always true. If $y \neq z$, one shows by lift reduction that then $b(y,z) = \bot$, hence the statement is trivial: We have:

$$\begin{aligned} & b(y,z) \\ = {} & \texttt{lift}_3\ (\lambda\,\alpha\ \beta\ \gamma.\ \alpha = x \wedge \beta = y \wedge \gamma = z)\ o\ o\ (\texttt{return}\ x) \\ = {} & \texttt{fmap}\ (\lambda\,\alpha.\ \alpha = x \wedge \alpha = y \wedge x = z)\ o \end{aligned}$$

The inner lambda term is `const False` unless $x = y = z$.

3: By 3.34.1, we have

$$z = (\texttt{const}\ z)() = (\texttt{return}\ ()) \rightsquigarrow (\texttt{const}\ z)$$

where the value () is the unique element of the unit type (). Now by axiom (*3.38), it suffices to show that

$$\forall x :: a, \zeta :: () : f\ x \preceq (\texttt{const}\ z)\ \zeta \quad \text{under } d \wedge o = x \wedge (\texttt{return}\ ()) = \zeta$$

Of course, there is only one possible value for ζ, namely (), so $(\texttt{return}\ () = \zeta) = \top$ and the statement above is equivalent to

$$\forall x :: a : f\ x \preceq_{d \wedge o = x} z$$

which is assumed to be true.

4: $f \approx (\texttt{const}\ x)$ means that for all $y :: a$, $f\ y \approx x$. Then apply 3. □

Remark 3.36. The converse of axiom (*3.38) now follows easily by lemma 3.35.2.

One receives the following "quantification theorem", which is not directly related to `read′`:

Corollary 3.37. *Let $x, y ::$ `Con`, $d ::$ `OB` and $o ::$ `Obs` a such that equality is lifted for a.*

If $x \preceq_{d \wedge o = \alpha} y$ for all $\alpha :: a$, then $x \preceq_d y$.

Note how this is essentially a case distinction over the usually infinitely many possible value of o while using the "normal" case distinction rule (*3.4), one can only consider finitely many cases.

Proof. We have $x \approx o \rightsquigarrow (\texttt{const}\ x)$ and $y \approx o \rightsquigarrow (\texttt{const}\ y)$ by lemma 3.35.4.

Now apply lemma 3.35.1. □

"$\rightsquigarrow$" commutes with the primitives which are time-local, i.e. all but `when′` and `anytime`, in the following sense:

Lemma 3.38. *Let a be a type such that equality is lifted for a, $o ::$ `Obs` a and $f, g :: a \to$ `Con`. Let $\alpha :: \mathbb{R}^+$. Then the following hold:*

$$\begin{aligned} (o \rightsquigarrow f) + (o \rightsquigarrow g) &\approx o \rightsquigarrow \lambda\, x.\ f\ x + g\ x \\ -(o \rightsquigarrow f) &\approx o \rightsquigarrow \lambda\, x.\ -(f\ x) \\ \alpha \cdot (o \rightsquigarrow f) &\approx o \rightsquigarrow \lambda\, x.\ \alpha \cdot f\ x \\ (o \rightsquigarrow f) \vee (o \rightsquigarrow g) &\approx o \rightsquigarrow \lambda\, x.\ f\ x \vee g\ x \end{aligned}$$

Proof. I only show the statement for `and`. The others follow in a similar way because they have similar monotonicity properties.

By lemma 3.35.3, it suffices to show that for all $x :: a$

$$(o \rightsquigarrow f) + (o \rightsquigarrow g) \approx_{o=x} f\ x + g\ x.$$

By lemma 3.35.2, $o \rightsquigarrow f \approx_{o=x} f\ x$ and the same holds for g, so by monotonicity of `and` (axiom (*3.6)), this is true. □

Remark 3.39. The above proof does not work for `when′` and `anytime`: The time when the observable is read matters and one needs to differentiate between acquisition time and the time where the condition becomes true (`when′`) / when the option is exercised (`anytime`).

Formally, the difference becomes visible by `when′` and `anytime` putting an additional condition into the premises of their monotonicity properties. E.g. for showing

$$\mathfrak{W}'\ b\ (o \rightsquigarrow f) \approx o \rightsquigarrow \lambda x.\ \mathfrak{W}'\ b\ (f\ x),$$

one would have to show that for all $x :: a$

$$\mathfrak{W}'\ b\ (o \rightsquigarrow f) \approx_{o=x} \mathfrak{W}'\ b\ (f\ x).$$

In contrast to the above combinators, this is *not* implied by $o \rightsquigarrow f \approx_{o=x} f\ x$ as $(o = x)$ is in general not in "$\mathfrak{e}$" form (cf. lemma 3.21).

This concludes the definition of the theory LPT.

3.2 Interim summary

Let's recap the structural properties of the primitives from the previous section:

- All primitives are monotonic (`give` is reversely monotonic) with respect to the relations "$\preceq_{\mathfrak{e}\ b}$". In particular, all primitives are compatible with "$\approx$".
- The primitives `and`, `give`, `scale` α and `read′` are even (reversely) monotonic with respect to all the relations "$\preceq_b$". I call these primitives *time-local*.
- All primitives commute with `zero`. The primitives `and`, `give`, `scale` α, `when′` and `read′` commute with `and`: They are group homomorphisms. I call these primitives *choice-free*.
- Anything commutes with `scale` α.
- `or` and `anytime` are uniquely defined by universal properties while `and`, `give`, `scale` and `read′` just expose certain known properties.

3.3 More about the structure of contracts

Remember how we defined

$$\texttt{when} :: \texttt{OB} \to \texttt{Con} \to \texttt{Con}$$
$$\texttt{when}\ b\ x = \mathfrak{n} \rightsquigarrow \lambda t.\ \mathfrak{W}'\ (b \wedge \mathfrak{n} \geq t)\ x.$$

`when` $b\ x =: \mathfrak{W}\ b\ x$ acquires x the *next* time b becomes true. Peyton Jones and Eber [3] introduced `when` instead of `when′` as a primitive while I chose `when′` for its simpler formal properties such as the collapse rule 3.22 and defined `when` in terms of `when′`.

One can also define `when′` in terms of `when` as the following lemma shows:

Lemma 3.40.

$$\mathfrak{W}'\ b\ x \approx \mathfrak{W}\ (\mathfrak{e}\ b)\ x$$

In particular, if $b = \mathfrak{e}\ b$, *then* $\mathfrak{W}'\ b\ x = \mathfrak{W}\ b\ x$.

Proof. One has to show:

$$\mathfrak{W}'\ b\ x \approx \mathfrak{n} \rightsquigarrow \lambda t.\ \mathfrak{W}'\ (\mathfrak{e}\ b \wedge \mathfrak{n} \geq t)\ x$$

Applying lemma 3.35.3 to the RHS, it suffices to show for all $t :: \texttt{Time}$ that

$$\mathfrak{W}'\ b\ x \approx_{\mathfrak{n}=t} \mathfrak{W}'\ (\mathfrak{e}\ b \wedge \mathfrak{n} \geq t)\ x.$$

By lemma 3.20.3, it is enough to show that

$$\mathfrak{e}\ (\mathfrak{n} = t) \Rightarrow (\mathfrak{e}\ b \leftrightarrow \mathfrak{e}\ (\mathfrak{e}\ b \wedge \mathfrak{n} \geq t)).$$

"$\leftarrow$": Even without the $\mathfrak{e}\ (\mathfrak{n} = t)$ precondition, $\mathfrak{e}\ b \Leftarrow (\mathfrak{e}\ b \wedge \mathfrak{n} \geq t)$, and so also $\mathfrak{e}\ b = \mathfrak{e}\ \mathfrak{e}\ b \Leftarrow \mathfrak{e}\ (\mathfrak{e}\ b \wedge \mathfrak{n} \geq t)$. (when in doubt, cf. lemma 2.19.4)

"$\rightarrow$": As by axiom (*2.2), $\mathfrak{e}\ (\mathfrak{n} = t) = \mathfrak{n} \geq t$, it suffices to show that $\mathfrak{n} \geq t \Rightarrow \ldots$. This is equivalent to

$$(\mathfrak{n} \geq t \wedge \mathfrak{e}\ b) \Rightarrow \mathfrak{e}\ (\mathfrak{e}\ b \wedge \mathfrak{n} \geq t),$$

which is clearly true. □

Remark 3.41. The above lemma is not limited to the $\mathfrak{n}$ observable: Let $\mathfrak{m} :: \texttt{Obs}\ a$ and define

$$f :: \texttt{OB} \to \texttt{Con} \to \texttt{Con}$$
$$f\ b\ x := \mathfrak{m} \rightsquigarrow \lambda u.\ \mathfrak{W}'\ (b \wedge \mathfrak{e}\ (\mathfrak{m} = u))\ x.$$

Then by the same proof as above one receives that

$$\mathfrak{W}'\ b\ x \approx f\ (\mathfrak{e}\ b)\ x.$$

3.3.1 Pricing lemma

By combining `scale` and "$\rightsquigarrow$", we receive the `money` function from figure 7:

$$\texttt{money} :: \texttt{Obs}\ \mathbb{R}^+ \to \texttt{Con}$$
$$\texttt{money}\ o = o \rightsquigarrow \lambda \alpha.\ \texttt{scale}\ \alpha\ \texttt{one}$$

Here, o is called the *price* of the contract `money` o (cf. section 4).

The following lemma is a generalization of lemma 3.12. It shows that prices must behave accordingly to the present value relations:

Lemma 3.42. *Let* $o, p :: \texttt{Obs}\ \mathbb{R}^+$. *Then the following conditions are equivalent, respectively, for any* $b :: \texttt{OB}$*:*

1. $\texttt{money}\ o \overset{\prec_b}{\underset{\preceq_b}{}}\ \texttt{money}\ p$.
2. $b \Rightarrow (o \overset{<}{\leq}\ p)$

Proof. First consider the "$\leq$" variant:

By axiom (*3.38) and lemma 3.35.2, 1 is equivalent to

$$\begin{aligned} & \forall \alpha, \beta : \alpha \cdot \texttt{one} \preceq_{b \wedge o = \alpha \wedge p = \beta} \beta \cdot \texttt{one}. \\ \Leftrightarrow\ & \forall \alpha, \beta : ((b \wedge o = \alpha \wedge p = \beta) = \bot) \vee \alpha \leq \beta \\ \Leftrightarrow\ & \forall \alpha, \beta : (b \wedge o = \alpha \wedge p = \beta) \Rightarrow (\alpha \leq \beta) \\ \Leftrightarrow\ & \forall \alpha, \beta : ((o, p) = (\alpha, \beta)) \Rightarrow (b \to \alpha \leq \beta) \end{aligned}$$

where the first equivalence is due to lemma 3.12 and the others are easily seen.

Now apply lemma 2.17.2 to see that this is equivalent to

$$\begin{aligned} \top &= ((o, p) \gg\!= \lambda\, (\alpha, \beta).\ b \to \alpha \leq \beta) \\ &= b \to o \leq p. \end{aligned}$$

Now show the "$<$" variant:

($1 \Rightarrow 2$): Assume that $b \not\Rightarrow o < p$, i.e. $d := (b \wedge o \geq p) \neq \bot$. As $d \Rightarrow o \geq p$, by the "$\leq$" part, $\texttt{money}\ o \succeq_d \texttt{money}\ p$. Also, $d \Rightarrow b$, so by definition of "$\prec_b$", $\texttt{money}\ o \not\prec_b \texttt{money}\ p$.

($2 \Rightarrow 1$): Assume that $\texttt{money}\ o \not\prec_b \texttt{money}\ p$. Then there is $d \Rightarrow b$, $d \neq \bot$, $\texttt{money}\ o \succeq_d \texttt{money}\ p$. By the "$\leq$" part, then $d \Rightarrow o \geq p$. So $b \not\Rightarrow o < p$. □

Corollary 3.43. *Lemma 3.42 still holds if one allows* $o, p :: \texttt{Obs}\ \mathbb{R}$ *and replaces* `money` *by* `moneyG`*.*

Proof. Just replace lemma 3.12 by corollary 3.13 in the above proof. □

3.4 Recursive equations for when′ and anytime

`when′` and `anytime` can be characterized by certain recursive equations. Define

$$\begin{aligned} \texttt{next} &:: \ \texttt{Con} \to \texttt{Con} \\ \texttt{next}\ x &:= \mathfrak{n} \rightsquigarrow \lambda\, t.\ \mathfrak{W}'\ (\mathfrak{n} > t)\ x. \end{aligned}$$

Of course, `next` x only matches its intuitive meaning when time is discrete: Then `next` x acquires x at the next point in time after its own acquisition time. If the assumption of discrete time is not made, `next` x can be thought of ignoring all effects of x at acquisition time. For example, if x is a `anytime` option, then `next` x is like x except for that the holder is not allowed to exercise immediately on acquisition.

Theorem 3.44. *Let* $x :: \texttt{Con}$ *and* $b :: \texttt{OB}$*. Then*

$$\mathfrak{W}'\ b\ x \approx \texttt{cond}\ (\mathfrak{e}\ b)\ x\ (\texttt{next}\ (\mathfrak{W}'\ b\ x)) \tag{3.39}$$

$$\mathfrak{A}\ x \approx x \vee \texttt{next}\ (\mathfrak{A}\ x) \tag{3.40}$$

Using the intuition for `next` from above, one receives the natural interpretations for (3.39) and (3.40):

- $\mathfrak{W}'\ b\ x$ is either x if $\mathfrak{e}\ b$ is true and otherwise nothing happens and the same check must be made again (strictly) later.
- $\mathfrak{A}\ x$ can be either exercised to receive x or otherwise nothing happens and one has the same choice again (strictly) later.

Proof. $\mathfrak{W}'$ part:

The statement is clear under conditions $\mathfrak{e}\ b$. So consider conditions $\neg\mathfrak{e}\ b$. We have

$$\begin{aligned}\texttt{cond}\ (\mathfrak{e}\ b)\ x\ (\texttt{next}\ (\mathfrak{W}'\ b\ x)) &\approx_{\neg\mathfrak{e}\ b} \texttt{next}\ (\mathfrak{W}'\ b\ x)\\ &= \quad \mathfrak{n} \rightsquigarrow \lambda t.\ \mathfrak{W}'\ (\mathfrak{n} > t)\ (\mathfrak{W}'\ b\ x)\\ &\approx \quad \mathfrak{n} \rightsquigarrow \lambda t.\ \mathfrak{W}'\ (\mathfrak{e}\ (\mathfrak{n} > t) \wedge \mathfrak{e}\ b)\ x.\end{aligned}$$

So it suffices to show that for any t

$$\mathfrak{W}'\ b\ x \approx_{\neg\mathfrak{e}\ b \wedge \mathfrak{n}=t} \mathfrak{W}'\ (\mathfrak{e}\ (\mathfrak{n} > t) \wedge \mathfrak{e}\ b)\ x.$$

I show that it holds even under conditions $\mathfrak{e}\ (\neg\mathfrak{e}\ b \wedge \mathfrak{n} = t)$. Via lemma 3.20.3 it suffices to show that

$$\mathfrak{e}\ (\neg\mathfrak{e}\ b \wedge \mathfrak{n} = t) \Rightarrow (\mathfrak{e}\ b \leftrightarrow \mathfrak{e}\ (\mathfrak{e}\ (\mathfrak{n} > t) \wedge \mathfrak{e}\ b)),$$

i.e. that

$$\mathfrak{e}\ (\neg\mathfrak{e}\ b \wedge \mathfrak{n} = t) \wedge \mathfrak{e}\ b \Rightarrow \mathfrak{n} > t.$$

This follows directly from lemma 2.25 with $b := \mathfrak{n} = t$ and $c := \neg\mathfrak{e}\ b$: We receive that the LHS implies $\bar{\mathfrak{e}}\ (\mathfrak{n} = t) = \mathfrak{n} > t$.

$\mathfrak{A}$ part:

"$\succeq$": We have $\mathfrak{A}\ x \succeq x$ and $\mathfrak{A}\ x \succeq \texttt{next}\ (\mathfrak{A}\ x)$ because for any $t :: \texttt{Time}$ we have $\mathfrak{A}\ x \succeq \mathfrak{W}'\ (\mathfrak{n} > t)\ (\mathfrak{A}\ x)$ by definition of $\mathfrak{A}$.

"$\preceq$": I show that $x \vee \texttt{next}\ (\mathfrak{A}\ x)$ fulfills the preconditions for the minimality axiom (*3.31): We have $x \vee \texttt{next}\ (\mathfrak{A}\ x) \succeq x$ trivially.

Let $b :: \texttt{OB}$. I show that

$$\mathfrak{W}'\ b\ (x \vee \texttt{next}\ (\mathfrak{A}\ x)) \preceq x \vee \texttt{next}\ (\mathfrak{A}\ x).$$

Under conditions $\mathfrak{e}\ b$, this is clear, so consider conditions $\neg\mathfrak{e}\ b$. We have for all $t :: \texttt{Time}$

$$\begin{aligned}\texttt{next}\ (\mathfrak{A}\ x) &\approx_{\mathfrak{n}=t} && \mathfrak{W}'\ (\mathfrak{n} > t)\ (\mathfrak{A}\ x)\\ &\succeq && \mathfrak{W}'\ (\mathfrak{n} > t)\ (\mathfrak{W}'\ b\ (\mathfrak{A}\ x))\\ &\approx && \mathfrak{W}'\ (\mathfrak{e}\ (\mathfrak{n} > t) \wedge \mathfrak{e}\ b)\ (\mathfrak{A}\ x)\\ &\succeq && \mathfrak{W}'\ (\mathfrak{e}\ (\mathfrak{n} > t) \wedge \mathfrak{e}\ b)\ (x \vee \texttt{next}\ (\mathfrak{A}\ x))\\ &\approx_{\neg\mathfrak{e}\ b \wedge \mathfrak{n}=t} && \mathfrak{W}'\ b\ (x \vee \texttt{next}\ (\mathfrak{A}\ x))\end{aligned}$$

where the last relation was seen in the first part of this proof and the others are applications of monotonicity of $\mathfrak{W}'$ and simple transformations. □

The converse of the previous theorem would be the statement that any contract a for which e.g. (3.40) holds (where $\mathfrak{A}\ x$ is replaced by a) is equal to `anytime`. But this is not clear in general:

For example, consider a model $\mathscr{A}$ where $\texttt{Time}^{\mathscr{A}}$ is the ordinal number $\omega + \omega$ and assume that this is known to the theory by – say – some constant symbols c_0 in the "lower part" and c_1 in the "upper part". If $a \approx x \vee \texttt{next}\ a$ is acquired at time c_0, then a models waiting any finite number of time steps by successively choosing the `next` a alternative, but it is not clear how one would model waiting for time c_1.

One can require that this situation does not occur as far as the theory is concerned by the following axiom:

Definition 3.45. *Reverse inductive time* is the following schema:

For any formula with parameters $\phi(t :: \texttt{Time}, \bar{y} :: \bar{a})$ we have the following:

$$\forall \bar{y} : (\forall t : (\forall t' > t : \phi(t', \bar{y})) \to \phi(t, \bar{y})) \to (\forall t : \phi(t, \bar{y}))$$

It is easy to see that under reverse inductive time the `Time` type is either empty or it has a maximum. Examples include any model where `Time` is finite and the ordering $\overleftarrow{\omega}$ of the natural numbers with their ordering reversed.

Under the assumption of reverse inductive time, we will see that the equations from theorem 3.44 are sufficient to characterize $\mathfrak{W}'\ b\ x$ and $\mathfrak{A}\ x$, respectively.

I first show a very helpful technical lemma which states that reverse induction can also be done via `next`:

Lemma 3.46. *Assume reverse inductive time. Let* $x, y :: \texttt{Con}$ *and* $d :: \texttt{OB}$ *be such that for all* $c :: \texttt{OB}$ *we have*

$$\texttt{next}\ x \preceq_{c \wedge \mathfrak{e}\ d} \texttt{next}\ y \Rightarrow x \preceq_{c \wedge \mathfrak{e}\ d} y.$$

Then $x \preceq_{\mathfrak{e}\ d} y$.

Proof. Wlog. assume that $d = \mathfrak{e}\ d$. Via corollary 3.37 it suffices to show that $x \preceq_{\mathfrak{n}=t \wedge d} y$ for any $t :: \texttt{Time}$. By assumption it now suffices to show by reverse induction that

$$\texttt{next}\ x \preceq_{\mathfrak{n}=t \wedge d} \texttt{next}\ y$$

for any t.

So fix a $t :: \texttt{Time}$ and assume that the statement holds for any $t' > t$. We have

$$\texttt{next}\ x \approx_{\mathfrak{n}=t} \mathfrak{W}'\ (\mathfrak{n} > t)\ x$$
$$\texttt{next}\ y \approx_{\mathfrak{n}=t} \mathfrak{W}'\ (\mathfrak{n} > t)\ y$$

and by monotonicity of $\mathfrak{W}'\ (\mathfrak{n} > t)$ it is enough to show that $x \preceq_{\mathfrak{n}>t \wedge d} y$. (Here, the condition $\mathfrak{n} > t \wedge d$ is of the right form for lemma 3.21 because $\mathfrak{n} > t = \mathfrak{e}\ (\mathfrak{n} > t)$ and $d = \mathfrak{e}\ d$ and so $\mathfrak{n} > t \wedge d = \mathfrak{e}\ (\mathfrak{n} > t \wedge d)$.)

It remains to see that this follows from the inductive assumption: By another application of corollary 3.37, the above statement is equivalent to having for all $t' :: \texttt{Time}$

$$x \preceq_{d \wedge \mathfrak{n}>t \wedge \mathfrak{n}=t'} y$$

which is trivially true for $t' \leq t$ and for $t' > t$ it is equivalent to

$$x \preceq_{d \wedge \mathfrak{n}=t'} y$$

which is given by the inductive assumption. □

Theorem 3.47. *Assume reverse inductive time. Let* $x :: \texttt{Con}$ *and* $b :: \texttt{OB}$.

1. *If* $w :: \texttt{Con}$ *is such that*

$$w \approx \texttt{cond}\ (\mathfrak{e}\ b)\ x\ (\texttt{next}\ w),$$

then $w \approx \mathfrak{W}'\ b\ x$.

2. *If* $a :: \texttt{Con}$ *is such that*

$$a \approx x \vee (\texttt{next}\ a),$$

then a *fulfills the universal property of* $\mathfrak{A}\ x$. *In particular,* $a \approx \mathfrak{A}\ x$.

Proof. 1: From (3.39) and the assumption on w it is clear that whenever $\texttt{next}\ w \approx_c \texttt{next}\ (\mathfrak{W}'\ b\ x)$, then $w \approx_c (\mathfrak{W}'\ b\ x)$. Hence, by lemma 3.46 for $d = \top$, the claim follows.

2: By the same argument as in the first part, it is clear that $a \approx \mathfrak{A}\ x$. I show directly that a must have the universal property, without using the existence of $\mathfrak{A}\ x$.

I show the axioms of $\mathfrak{A}\ x$, where $\mathfrak{A}\ x$ is replaced with a:

$$a \succeq x \qquad (*3.29/a)$$

$$\forall b :: \texttt{OB} :\ a \succeq \mathfrak{W}'\ b\ a \qquad (*3.30/a)$$

$$(z \succeq_{\mathfrak{e}\ d} x \text{ and } \forall b :: \texttt{OB} :\ z \succeq_{\mathfrak{e}\ d} \mathfrak{W}'\ b\ z) \Rightarrow z \succeq_{\mathfrak{e}\ d} a \qquad (*3.31/a)$$

(*3.29/a) is clear.

(*3.30/a) can be shown using lemma 3.46: Assume that $\texttt{next}\ a \succeq_c \texttt{next}\ (\mathfrak{W}'\ b\ a)$. Now:

$$\begin{aligned} a \approx a \vee \texttt{next}\ a &\succeq_c a \vee \texttt{next}\ (\mathfrak{W}'\ b\ a) \\ &\succeq \texttt{cond}\ (\mathfrak{e}\ b)\ a\ (\texttt{next}\ (\mathfrak{W}'\ b\ a)) \\ &\approx \mathfrak{W}'\ b\ a. \end{aligned}$$

Here, the middle equation is just because $a \vee \texttt{next}\ (\mathfrak{W}'\ b\ a)$ is greater than each of the two branches of the `cond` and the last one is (3.39).

(*3.31/a) follows again using lemma 3.46: Assume that z is as in (*3.31/a) and assume that $\texttt{next}\ a \preceq_{\mathfrak{e}\ d \wedge c} \texttt{next}\ z$ for some $c :: \texttt{OB}$.

We have $\texttt{next}\ z \preceq_{\mathfrak{e}\ d} z$ (as $\mathfrak{W}'\ (\mathfrak{n} > t)\ z \preceq_{\mathfrak{e}\ d} z$ for all t) and $x \preceq_{\mathfrak{e}\ d} z$. So

$$z \succeq_{\mathfrak{e}\ d} x \vee \texttt{next}\ z \succeq_{\mathfrak{e}\ d \wedge c} x \vee \texttt{next}\ a \approx a.$$ □

Remark 3.48. The proof of the $\mathfrak{A}\ x$ part of the previous lemma did not use the existence of $\mathfrak{A}\ x$. Hence, when constructing a model, it is enough to show that the model for $\mathfrak{A}\ x$ has the property 3.47.2 in order to see that the axioms for $\mathfrak{A}\ x$ are satisfied. This fact will be exploited in section 5.3.

For `when′`, no such variant can be given: The definition of `next` already uses `when′`.

4 Applications

In this section, I present formalizations in LPT of the fundamental concepts from finance such as prices and interest and I give formal proofs of the best known theorems from arbitrage theory.

Informal proofs of all statements can be found in [1]. Hull describes arbitrage statements for *dividend-free shares* such as the famous put-call parity and the rule for forward prices. One of the core questions this section is going to answer is what a dividend-free share is actually supposed to *be*.

4.1 Prices

The notion of a price is typically treated by common sense, but it is easy to define formally in LPT:

Definition 4.1. If $o :: \texttt{Obs}\ \mathbb{R}$, $x :: \texttt{Con}$ and $b :: \texttt{OB}$, then o is called a *price* for x (under conditions b) if $x \approx \texttt{moneyG}\ o$ ($x \approx_b \texttt{moneyG}\ o$).

Note that prices are unique by corollary 3.43 if they exist. Whether or not prices exist is not clear: In the discrete-time model from section 5.3 they always exist, but example 4.14 describes a contract which cannot have a price if dense time is assumed.

Remark 4.2. As usual, I left out currencies. It is clear that a price is always coupled with the currency it is denoted in. Different currencies will be considered in section 4.3.

One receives that prices for the time-local combinators can easily be computed, so the difficult points are really only $\texttt{when}'$ and `anytime`:

Theorem 4.3. *Let $x, y :: \texttt{Con}$ and let $o, p :: \texttt{Obs}\ \mathbb{R}$ be prices for x and y, respectively.*

1. $\texttt{return}\ 0$ *is a price for* `zero`.
2. $-o$ *is a price for* $\texttt{give}\ x$.
3. $o + p$ *is a price for* $\texttt{and}\ x\ y$.
4. $\max(o, p)$ *is a price for* $\texttt{or}\ x\ y$.
5. *If $q :: \texttt{Obs}\ a$, $f :: a \to \texttt{Con}$ and $g :: a \to \texttt{Obs}\ \mathbb{R}$ are such that for any $x :: a$, $g\ x$ is a price for $f\ x$ under conditions $(o = x)$, then $q \gg\!= g$ is a price for $q \rightsquigarrow f$.*

These statements extend to prices under conditions as well.

Proof. Part 5: We have

$$\begin{aligned} q \rightsquigarrow f &\approx q \rightsquigarrow \lambda x.\, \texttt{moneyG}\ (g\ x) \\ &= q \rightsquigarrow \lambda x.\, g\ x \rightsquigarrow \lambda \alpha.\, \texttt{scaleG}\ \alpha\ \texttt{one} \\ &\approx (q \gg\!= g) \rightsquigarrow \lambda \alpha.\, \texttt{scaleG}\ \alpha\ \texttt{one} \\ &= \texttt{moneyG}\ (q \gg\!= g) \end{aligned}$$

where the first relation follows from the assumption and the third is lemma 3.34.3.

Other parts: As all the combinators are compatible with "$\approx$", it suffices to show the following:

$$\begin{aligned}
\texttt{moneyG}\ (\texttt{return}\ 0) &\approx \texttt{zero}\\
\texttt{moneyG}\ (-o) &\approx \texttt{give}\ (\texttt{moneyG}\ o)\\
\texttt{moneyG}\ (o+p) &\approx \texttt{and}\ (\texttt{moneyG}\ o)\ (\texttt{moneyG}\ p)\\
\texttt{moneyG}\ (\max(o,p)) &\approx \texttt{or}\ (\texttt{moneyG}\ o)\ (\texttt{moneyG}\ p)
\end{aligned}$$

It is easy to see using the properties of "$\rightsquigarrow$" from section 3.1.8 that it now suffices to show the following block:

$$\begin{aligned}
\texttt{scaleG}\ 0\ \texttt{one} &\approx \texttt{zero}\\
\texttt{scaleG}\ (-\alpha)\ \texttt{one} &\approx \texttt{give}\ (\texttt{scaleG}\ \alpha\ \texttt{one})\\
\texttt{scaleG}\ (\alpha+\beta)\ \texttt{one} &\approx \texttt{and}\ (\texttt{scaleG}\ \alpha\ \texttt{one})\ (\texttt{scaleG}\ \beta\ \texttt{one})\\
\texttt{scaleG}\ (\max(\alpha,\beta))\ \texttt{one} &\approx \texttt{or}\ (\texttt{scaleG}\ \alpha\ \texttt{one})\ (\texttt{scaleG}\ \beta\ \texttt{one})
\end{aligned}$$

To see this, note how the four combinators are time-local and hence are compatible with relations such as "$\approx_{o=\alpha}$". This would already fail for $\texttt{when}'\ b$ and `anytime`!

Now the equations for `zero` and `give` are clear by definition of `scaleG`. For `and` and `or`, the relations were were seen in axiom (*3.17) and lemma 3.17, respectively, if one replaces `scaleG` with `scale`. The `scaleG` variants are then seen using a simple case distinction.

All arguments extend to prices under conditions because all time-local combinators are compatible with all the "$\approx_b$" relations. □

4.2 Interest

A essential concept related to prices are (risk-free) interest rates which describe the "price of future money".

Definition 4.4. Define $\overline{\mathbb{R}}^+ = \mathbb{R}^+ \cup \{\infty\}$ where ∞ should have the usual informal semantics like $\frac{1}{\infty} = 0$ etc. $\overline{\mathbb{R}}^+$ could be modeled by $\texttt{Maybe}\ \mathbb{R}^+$ where `Nothing` represents ∞ and many case distinctions.

The (risk-free) *zero-coupon bond* (*ZCB*) for $K :: \mathbb{R}^+$ and $b :: \texttt{OB}$ is the contract K^b that pays K dollars as soon as b becomes true. I.e.:

$$K^b := \texttt{when}'\ b\ (K \cdot \texttt{one})$$

If 1^b has a price $o :: \texttt{Obs}\ \mathbb{R}$, then $o \geq 0$ because $1^b \succeq 0$. Then define the (risk-free) *b-interest factor* $R_b :: \texttt{Obs}\ \overline{\mathbb{R}}^+$ as $R_b := \frac{1}{o}$. Note that $R_b > 0$ if it exists.

If $\Delta t :: \texttt{TimeDiff}$, $\Delta t \geq 0$, also write

$$K^{\Delta t} := \texttt{after}\ \Delta t\ (K \cdot \texttt{one}).$$

If $1^{\Delta t}$ has a price $o :: \texttt{Obs}\ \mathbb{R}$, then one receives analogously to above the Δt-*interest factor* $R_{\Delta t}$. In addition, one may define the Δt-*interest rate*[27]

$$r_{\Delta t} := \mathfrak{n} \rightsquigarrow \lambda t.\ R_{\Delta t}^{\frac{1}{\iota\ (t+\Delta t)-\iota\ t}} - 1.$$

[27] An alternative form would be $(\ln\ (\iota\ (t+\Delta t) - \iota\ t))/r_{\Delta t}$, so $R_{\Delta t} = \mathfrak{n} \rightsquigarrow \lambda t.\ \exp\ (r_{\Delta t} \cdot (\iota\ (t+\Delta t) - \iota\ t))$, the traditional form of continuous compound interest [1, sec. 4.2].

The function $\iota :: \texttt{Time} \to \mathbb{R}$ is from section A.3.3. In this section, it is also explained why there is not in general an embedding $\texttt{TimeDiff} \to \mathbb{R}$.

Clearly, interest factors and rates are unique if they exist. It is further clear that interest factors are the same if the face value 1 of the ZCB is replaced by anything else.

Note how we have

$$R_{\Delta t} = \mathfrak{n} \rightsquigarrow \lambda\, t.\ (1 + r_{\Delta t})^{\iota\ (t+\Delta t) - \iota\ t}$$

as expected. This could be written $R_{\Delta t} = (1 + r_{\Delta t})^{\Delta t}$ as well.

The above definition allows infinite interest rates and factors to model events that might not occur: An extreme example is $R_\perp$, a less extreme one is $R_{\Delta t}$ in a finite-time model in a situation where one is closer than Δt to the end of time: In both cases, a ZCB would never be paid, hence its price is 0. Thus, the respective interest factor and rate must be ∞. One could also argue that since the ZCB is never paid, writers can promise arbitrarily high – and hence "in price" infinite – interest rates.

Note that the idea of "interest" is generalized here: Typically, only the Δt-variants would be called actual "interest" constructions. The b-variants generalize the concept to uncertain events.

Remark 4.5. $r_{\Delta t}$ is well-defined:

- Recap that $t + \Delta t$ is actually $\texttt{timeOffset}\ t\ \Delta t :: \texttt{Maybe Time}$. The `Nothing` case is not considered above, but one can set $r_{\Delta t}$ to ∞ in situations where $\mathfrak{n} + \Delta t = \texttt{Nothing}$. This is reasonable by the following argument:

  Assume that $t + \Delta t = \texttt{Nothing}$. Then $\texttt{after}\ \Delta t\ (1 \cdot \texttt{one}) \approx_{\mathfrak{n}=t} 0$, hence $(\mathfrak{n} = t) \Rightarrow (R_{\Delta t} = \infty)$, and $\infty^\alpha = \infty$ for any $\alpha > 0$. Also, $\Delta t > 0$ because $t + 0 = \texttt{Just}\ t$.

- If $\iota\ (t + \Delta t) - \iota\ t = 0$, then $\Delta t = 0$ as both `timeOffset` and ι are strictly monotonic. It is easy to see that then $R_0 = 1$ and $1^\infty = 1$. So one has $R_0 = 1$.

It is usually assumed that the interest rates $r_{\Delta t}$ or interest factors $R_{\Delta t}$ always exist, but I shall not need this in general.

It is also often assumed that $R_{\Delta t}$ is a constant of form $R_{\Delta t} = \texttt{return}\ r$, $r \in \overline{\mathbb{R}}^+$, or that $r_{\Delta t}$ does not depend on Δt. The first assumption would mean that interest rates can't change over time. The second would mean that the yield curve is perfectly flat.[28] Both assumptions turn out to be wrong in practice and are not required here.

Another common assumption is the following.

Definition 4.6. *Non-negative interest rates* is the assumption that

$$\texttt{one} \approx \mathfrak{A}\ \texttt{one}.$$

Lemma 4.7. *The following are equivalent:*

1. *Non-negative interest rates hold.*

[28] Cf. "Term structure of interest rates" in [1]

2. $\texttt{one} \succeq \texttt{when}'\ b\ \texttt{one}$ *for any* $b :: \texttt{OB}$.

3. $\texttt{one} \succeq 1^b$ *for any* $b :: \texttt{OB}$.

Assume that all b-interest factors exist. Then the following is also equivalent to the previous group of statements.

4. $R_b \geq 1$ *for any* $b :: \texttt{OB}$.

Given that all Δt-interest factors exist, the previous group implies the following, which are also equivalent:

5. $\texttt{one} \succeq \texttt{when}'\ (\mathfrak{n} = t)\ \texttt{one}$ *for any* $t :: \texttt{Time}$.

6. $R_{\Delta t} \geq 1$ *for any* $\Delta t :: \texttt{TimeDiff}$.

7. $r_{\Delta t} \geq 0$ *for any* $\Delta t :: \texttt{TimeDiff}$.

Part 7 is what is typically meant when the term "non-negative interest rates" is used in an informal context.

Proof. All statements follow directly from the definitions. The transition from "$\preceq$" to comparing prices is done via corollary 3.43. □

One receives that the notion of an "interest rate" matches the intuition as follows:

Theorem 4.8. *Assume that* $b :: \texttt{OB}$, $\Delta t \geq 0$ *and their interest factors exist. Then*

$$R_b \rightsquigarrow \lambda\, \alpha.\ \mathfrak{W}'\ b\ \alpha \approx_{R_b < \infty} \texttt{one}$$
$$R_{\Delta t} \rightsquigarrow \lambda\, \alpha.\ \texttt{after}\ \Delta t\ \alpha \approx_{R_{\Delta t} < \infty} \texttt{one}.$$

Here, $\alpha = \infty$ may stand for any contract and $\alpha < \infty$ stands for $\alpha \cdot \texttt{one}$.

Proof. We have by definition $\frac{1}{R_b} \approx \mathfrak{W}'\ b\ \texttt{one}$. This implies that

$$\begin{aligned} \texttt{one} &\approx_{R_b < \infty} R_b \cdot \mathfrak{W}'\ b\ \texttt{one} \\ &= \quad R_b \rightsquigarrow \lambda\, \alpha.\ \alpha \cdot \mathfrak{W}'\ b\ \texttt{one} \\ &\approx \quad R_b \rightsquigarrow \lambda\, \alpha.\ \mathfrak{W}'\ b\ \alpha \end{aligned}$$

where the last relation is by axiom (*3.22).

The statement for Δt is shown analogously. □

Corollary 4.9. *Let b and Δt be as in theorem 4.8. Then*

$$R_b \rightsquigarrow \lambda\, \alpha.\ \mathfrak{W}'\ b\ \alpha \approx \texttt{cond}\ (R_b < \infty)\ \texttt{one}\ \texttt{zero}$$
$$R_{\Delta t} \rightsquigarrow \lambda\, \alpha.\ \texttt{after}\ \Delta t\ \alpha \approx \texttt{cond}\ (R_{\Delta t} < \infty)\ \texttt{one}\ \texttt{zero}$$

Here, $\alpha = \infty$ can stand for any contract of form $\beta \cdot \texttt{one}$, $\beta \in \mathbb{R}^+$.

Proof. The $(R_b < \infty)$ branch follows from theorem 4.8. For the other branch, we have that $\mathfrak{W}'\ b\ \texttt{one} \approx_{R_b = \infty} \texttt{zero}$ because $\frac{1}{R_b}$ is a price for $\mathfrak{W}'\ b\ \texttt{one}$ and then also $\mathfrak{W}'\ b\ \beta \approx_{R_b = \infty} \texttt{zero}$ for any $\beta \in \mathbb{R}^+$. □

A very simple consequence is that *fixed* future payments can be converted into payments at acquisition time by dividing by the respective interest factor:

Corollary 4.10 (Discounting future payments)**.** *If* $\alpha :: \mathbb{R}$, $b ::$ `OB` *and* R_b *exists, then*

$$\mathfrak{W}'\ b\ \alpha \approx \frac{1}{R_b} \cdot \alpha$$

Proof. Clear by the above. Note how the ∞ case works as well. □

Remark 4.11. The natural generalization of corollary 4.10 to `Obs` $\mathbb{R}$ is

$$o \rightsquigarrow \lambda\, \alpha.\ \mathfrak{W}'\ b\ \alpha \approx \frac{1}{R_b} \cdot o.$$

The LHS *cannot* be replaced by

$$\mathfrak{W}'\ b\ o.$$

The difference here is the point in time when the observable o is read: The former contract reads it at acquisition time while the latter reads it at time b. It is clear that this is not the same, for example for $o =$ `if'` (`c` b) 0 1.

The equations from theorem 4.8 – with the special interpretation for ∞ – uniquely define R_b:

Lemma 4.12. *Let* $b ::$ `OB` *and let* $o ::$ `Obs` $\overline{\mathbb{R}}^+$ *be such that*

$$\begin{aligned} \mathfrak{W}'\ b\ \beta &\approx_{o=\infty} \texttt{zero} \qquad \text{for all } \beta \in \mathbb{R}^+ \\ o \rightsquigarrow \lambda\, \alpha.\ \mathfrak{W}'\ b\ \alpha &\approx_{o<\infty} \texttt{one}. \end{aligned}$$

Then $\frac{1}{o}$ *is a price for* $\mathfrak{W}'\ b$ `one`*, i.e.* $o = R_b$*.*

The respective statement for Δt *holds as well.*

Proof. First consider conditions $o = \infty$, which implies $\frac{1}{o} = 0$. So one needs to show that $\mathfrak{W}'\ b$ `one` $\approx_{o=\infty}$ `zero`, which is true by the first condition for $\beta = 1$.

Now consider conditions $o < \infty$. Then the statement follows by multiplying by $\frac{1}{o}$ like in the proof of theorem 4.8. □

Remark 4.13 (Negative interest rates). At first sight, the assumption that

$$\texttt{one} \succeq \mathfrak{W}'\ b\ \texttt{one}$$

seems reasonable: A trader having a dollar, i.e. `one`, in cash can commit to not use the money before b, therewith transforming it into $\mathfrak{W}'\ b$ `one`. The dollar is stored until time b.

However, it was seen in lemma 4.7 that this statement is equivalent to non-negative interest rates (also from an intuitive point of view, it is also clear that an interest rate of 0 can always be realized by storing money in cash) and we know that negative interest rates, though rare, have happened in the past.

It follows that the intuition of `one` as "cash" must be wrong and there can be a cost associated to "storing" money.

Example 4.14 (A priceless contract). Assume that time is dense in the following sense:

$$\begin{aligned}\forall t :: \mathtt{Time}, \varepsilon :: \mathbb{R}^+, \varepsilon > 0\ \exists \Delta t :: \mathtt{TimeDiff}, t' :: \mathtt{Time} :\\ t + \Delta t = \mathtt{Just}\ t' \wedge 0 < \iota\ t' - \iota\ t < \varepsilon\end{aligned}$$

Assume further that all Δt-interest rates exist and that there is an observable $o :: \mathtt{Obs}\ \mathbb{R}^+$ such that $r_{\Delta t} \leq o$ for any Δt. Define for $t :: \mathtt{Time}$ the following observables:

$$\begin{aligned}a\ t &:= \mathfrak{n} \rightsquigarrow \lambda t'.\ (1+o)^{t'-t}\\ b\ t &:= \mathfrak{n} \rightsquigarrow \lambda t'.\ \mathtt{if'}\ (t = t')\ 0\ \left(\frac{1}{t'-t}\right)\end{aligned}$$

Here, $t' - t$ is short for $\iota\ t' - \iota\ t$. Then the following contract has no price:

$$x := \mathfrak{n} \rightsquigarrow \lambda t.\ \mathfrak{A}\ (a\ t \cdot b\ t)$$

The idea is that the b part of x pays arbitrarily high amounts if the holder only exercises quickly enough strictly after acquisition. The factor a is used to equate for the time difference between acquisition and exertion time. It is only needed in the pathological situation where the $r_{\Delta t}$ values behave similarly.

As it can't have a reasonable price, the contract x would not be traded at a stock exchange. It is however possible that a "priceless" derivative is exchanged for another one in a direct (or "over-the-counter") agreement between two traders.

Proof. I show that $x \succeq \alpha$ for any $\alpha :: \mathbb{R}$. Then also $x \succeq \alpha + 1 \succ \alpha$ for any α and so $x \succ o$ for any $o :: \mathtt{Obs}\ \mathbb{R}$, hence there is no price.

So let $\alpha > 0$. Let $t :: \mathtt{Time}$ and choose Δt and t' as in the assumption for $\varepsilon = \frac{1}{\alpha}$.

We have

$$\begin{aligned}\alpha = &\quad \alpha \cdot \mathtt{one}\\ \approx_{\mathfrak{n}=t} &\ (1 + r_{\Delta t})^{t'-t} \cdot \mathtt{after}\ \Delta t\ \alpha\\ \preceq &\quad o^{t'-t} \cdot \mathtt{after}\ \Delta t\ \alpha\\ \approx_{\mathfrak{n}=t} &\ \mathtt{after}\ \Delta t\ (a\ t \cdot \alpha).\end{aligned}$$

The last relation is easily seen from the facts that $\mathtt{after}\ \Delta t\ y \approx_{\mathfrak{n}=t} \mathfrak{W}'\ (\mathfrak{n} = t')\ y$ and $(\mathfrak{n} = t') \Rightarrow a\ t = o^{t'-t}$.

We further have $(\mathfrak{n} = t') \Rightarrow b\ t = \frac{1}{t'-t} \geq \alpha$. So in total we get

$$\begin{aligned}\alpha \preceq_{\mathfrak{n}=t} &\ \mathtt{after}\ \Delta t\ (a\ t \cdot b\ t)\\ \preceq &\quad \mathfrak{A}\ (a\ t \cdot b\ t) = x.\end{aligned}$$

As t was arbitrary, the claim follows (using corollary 3.37). □

Remark 4.15. In the previous proof, it would have been sufficient to assume that $r_{\Delta t} \leq o$ for sufficiently small Δt. This assumption is reasonable as $r_{\Delta t}$ is typically decreasing as Δt decreases, at least for small Δt.

4.3 Exchange Rates

Again related to prices is the "price of another currency", i.e. the exchange rate.

Definition 4.16. Let $k, s :: \texttt{Currency}$. The *$k/s$-exchange rate* $W^{k/s} :: \texttt{Obs}\ \mathbb{R}^+$ – if it exists – is the price of $\texttt{one}\ s$ in currency k, i.e.

$$\texttt{one}\ s \approx W^{k/s} \cdot \texttt{one}\ k.$$

The definition extends to include market conditions of type `OB` as usual. By uniqueness of prices, exchange rates are unique if they exist.

Theorem 4.17 (Currency conversion / cross rates)**.** *Let $k, s :: \texttt{Currency}$ and assume that $W^{k/s}$ exists.*

1. *$W^{k/k}$ always exists and $W^{k/k} = \texttt{return}\ 1$*
2. *$W^{k/s} > 0$*
3. *$W^{s/k}$ exists as well and $W^{s/k} = \frac{1}{W^{k/s}}$*
4. *If $x :: \texttt{Con}$ and $o :: \texttt{Obs}\ \mathbb{R}$ is a price for x in currency s, then $o \cdot W^{k/s}$ is a price for x in currency k.*
5. *If $t :: \texttt{Currency}$ is such that $W^{s/t}$ exists, then $W^{k/t}$ exists as well and*

$$W^{k/t} = W^{k/s} \cdot W^{s/t}. \tag{4.1}$$

The statements all extend to prices under conditions.

Equation (4.1) is typically called *cross rates equation.*

Proof. 1: Clearly, $\texttt{return}\ 1$ is a price for k in currency k.
2: We have

$$0 \cdot \texttt{one}\ k \approx \texttt{zero} \prec \texttt{one}\ s \approx W^{k/s} \cdot \texttt{one}\ k.$$

Then by lemma 3.42 the claim follows.
3:

$$\begin{aligned} &\frac{1}{W^{k/s}} \cdot \texttt{one}\ s \approx \frac{1}{W^{k/s}} \cdot \left(W^{k/s} \cdot \texttt{one}\ k\right) \\ \approx &\left(\frac{1}{W^{k/s}} \cdot W^{k/s}\right) \cdot \texttt{one}\ k \approx 1 \cdot \texttt{one}\ k \\ \approx &\ \texttt{one}\ k \end{aligned}$$

The "associativity" of "$\cdot$" used in the middle relation is a straightforward generalization of axiom (*3.13) to $\texttt{Obs}\ \mathbb{R}$.
4:

$$\begin{aligned} x &\approx o \cdot (\texttt{one}\ s) \\ &\approx o \cdot \left(W^{k/s} \cdot \texttt{one}\ k\right) \\ &\approx \left(o \cdot W^{k/s}\right) \cdot \texttt{one}\ k \end{aligned}$$

5: Apply part 4 to $x = \texttt{one}\ t$ and $o = W^{s/t}$. Then $W^{k/s} \cdot W^{s/t}$ is a price for $\texttt{one}\ t$ in currency k as wanted. □

Remark 4.18.

1. It is usually assumed that all exchange rates exist. If this assumption is dropped, one receives the classes of the equivalence relation "have an exchange rate" as separate commodities which are not directly related.

2. Foreign exchange is one of the places where one typically hits the *BID-ASK spread* in reality: The rate at which one can buy, say, USD for EUR (ASK) is typically lower than the rate for which one can sell USD for EUR (BID), so going from EUR to USD and back again results in a loss. Cf. section 6.1.

4.4 Forwards

After the many different flavors of prices, one can consider the first example of a "real" derivative, namely a *forward*. Forward contracts fix a price for a certain asset in advance:

Definition 4.19. A *forward contract* allows and obliges the holder to buy a certain asset at a certain point in the future for a fixed price K. The position described here is called *long forward*, the counterparty position is called *short forward*.

Formally:

$$\begin{aligned} \texttt{forward} &:: \ \texttt{OB} \to \mathbb{R} \to \texttt{Con} \to \texttt{Con} \\ \texttt{forward}\ b\ K\ x &:= \mathfrak{W}'\ b\ (x - K) \end{aligned}$$

Also write $\mathrm{F}_{b,K,x}$ for $\texttt{forward}\ b\ K\ x$. For $o :: \texttt{Obs}\ \mathbb{R}^+$ also write $\mathrm{F}_{b,o,x}$ and $\texttt{forward}\ b\ o\ x$ for

$$o \rightsquigarrow \lambda K.\ \mathrm{F}_{b,K,x}.$$

Recap that this is completely different from $\mathfrak{W}'\ b\ (x - o)$: The price K is fixed at acquisition time, not at time b.

$o :: \texttt{Obs}\ \mathbb{R}$ is called a *forward price* for x and b (under conditions c) if $\mathrm{F}_{b,o,x} \approx 0$ ($\mathrm{F}_{b,o,x} \approx_c 0$).

Usually in the above definition, $b = (\mathfrak{n} = T)$ for some $T :: \texttt{Time}$, but I shall not be so strict for the general case.

Theorem 4.20 (Forward Prices)**.** *Let $x :: \texttt{Con}$ and $b :: \texttt{OB}$. Let $o :: \texttt{Obs}\ \mathbb{R}$ be a price for x and assume that the interest factor R_b exists. The following statements are equivalent:*

1. *$R_b \cdot o$ is a forward price for x and b under conditions $R_b < \infty$.*

2. *$\mathfrak{W}'\ b\ x \approx_{R_b < \infty} x$.*

Here, the ∞ value of R_b can be assigned any contract, as usual.

Proof.

$$\begin{aligned} \mathrm{F}_{b,(R_b \cdot o),x} = \quad & (R_b \cdot o) \rightsquigarrow \lambda K.\ \mathfrak{W}'\ b\ (x - K) \\ \approx \quad & \mathfrak{W}'\ b\ x - (R_b \cdot o) \rightsquigarrow \lambda K.\ \mathfrak{W}'\ b\ K \\ \approx_{R_b < \infty} \quad & \mathfrak{W}'\ b\ x - o \\ \approx \quad & \mathfrak{W}'\ b\ x - x \end{aligned}$$

The last relation is because $x \approx o$ by assumption and the previous one is by corollary 4.10. □

The above equivalence gives a first indication for how to define a dividend-free share: Hull[1, sec. 5.4] states that statement 1 should hold whenever $b = (\mathfrak{n} = T)$.

Corollary 4.21 (Forward Exchange Rate)**.** *Let* $k, s :: \texttt{Currency}$*,* $b :: \texttt{OB}$ *and assume that* $W^{k/s}$*,* $R_{b,k}$ *and* $R_{b,s}$ *exist (interest factors for the two currencies). Then*

$$W^{k/s} \cdot \frac{R_{b,k}}{R_{b,s}}$$

is a forward price in currency k *for* b *and* `one` s *under conditions* $R_{b,k} < \infty \wedge R_{b,s} < \infty$*.*

This observable is also called k/s-forward exchange rate *for* b*.*

Proof. Let $x = \frac{1}{R_{b,s}} \cdot \texttt{one}\ s$. We have that $\mathfrak{e}\ b \Rightarrow R_{b,s} = 1$ and hence $x \approx \mathfrak{W}'\ b\ (\texttt{one}\ s) \approx \mathfrak{W}'\ b\ x$, both by definition of $R_{b,s}$.

$\frac{1}{R_{b,s}}$ is trivially a price for x in currency s and hence by lemma 4.17.4, $o := W^{k/s} \cdot \frac{1}{R_{b,s}}$ is a price for x in currency k.

Now apply the previous theorem 4.20 in currency k to x and o to receive that $R_{b,k} \cdot o = W^{k/s} \cdot \frac{R_{b,k}}{R_{b,s}}$ is a forward price for x in currency k. Finally, note again that $x \approx_{\mathfrak{e}\ b} \texttt{one}\ s$, so by definition of `forward`, any forward price for x is also a forward price for `one` s. □

4.5 European options, put-call parity

If one removes from forwards the obligation of the holder to buy even if it is unprofitable for her, one arrives at European *options*. There will be no equivalent of the forward price for options.

Definition 4.22. A *European call option* is a contract that grants the holder the right, but not the obligation, to buy a certain asset at a certain point in the future for a fixed price K. A *European put option* grants the holder to sell the asset for a fixed price.

In general terms, define

$$\begin{aligned} \texttt{europeanCall} &:: \texttt{OB} \to \mathbb{R}^+ \to \texttt{Con} \to \texttt{Con} \\ \texttt{europeanCall}\ b\ K\ x &= \mathfrak{W}'\ b\ ((x - K) \vee 0) \end{aligned}$$

$$\begin{aligned} \texttt{europeanPut} &:: \texttt{OB} \to \mathbb{R}^+ \to \texttt{Con} \to \texttt{Con} \\ \texttt{europeanPut}\ b\ K\ x &= \mathfrak{W}'\ b\ ((K - x) \vee 0) \end{aligned}$$

Also write $\mathrm{C}_{b,K,x}$ for $\texttt{europeanCall}\ b\ K\ x$ and $\mathrm{P}_{b,K,x}$ for $\texttt{europeanPut}\ b\ K\ x$.

As for forwards, one often uses $b = (\mathfrak{n} = T)$ for some $T :: \texttt{Time}$.

One can now continue to show the put-call parity from section 1.1 formally:

Definition 4.23. Recap the definition of K^b from section 4.2.

A triple $(x :: \texttt{Con}, b :: \texttt{OB}, K :: \mathbb{R}^+)$ *satisfies the put-call parity* (*PCP*) if

$$\mathrm{P}_{b,K,x} + x \approx \mathrm{C}_{b,K,x} + K^b \tag{PCP}$$

Theorem 4.24. *Let* $x :: \texttt{Con}$ *and* $b :: \texttt{OB}$*. The following are equivalent:*

1. (x, b, K) *satisfies the PCP for some* $K :: \mathbb{R}^+$*.*
2. (x, b, K) *satisfies the PCP for all* $K :: \mathbb{R}^+$*.*
3. $\mathfrak{W}'\ b\ x \approx x$*.*

Proof. $(2 \Rightarrow 1)$ is trivial.

$(1 \Rightarrow 3 \Rightarrow 2)$: I show that for all K, (x, b, K) satisfies the PCP iff $\mathfrak{W}'\ b\ x \approx x$. (PCP) is equivalent to

$$\begin{aligned} x &\approx \mathrm{C}_{b,K,x} + -\mathrm{P}_{b,K,x} + K^b \\ &= \mathfrak{W}'\ b\ ((x - K) \vee 0) - \mathfrak{W}'\ b\ ((K - x) \vee 0) + \mathfrak{W}'\ b\ K \\ &\approx \mathfrak{W}'\ b\ (((x - K) \vee 0) - ((K - x) \vee 0) + K). \end{aligned}$$

So it suffices to show that the inner contract on the RHS is equal in present value to x.

To see that, note that

$$\begin{aligned} (x - K) \vee 0 &\approx (x \vee K) - K \\ (K - x) \vee 0 &\approx (K \vee x) - x \end{aligned}$$

by lemma 3.15.1. So

$$\begin{aligned} &(((x - K) \vee 0) - ((K - x) \vee 0) + K \\ \approx\ &((x \vee K) - K) - ((K \vee x) - x) + K \\ \approx\ &(x \vee K) - K - (K \vee x) + x + K \\ \approx\ &x. \end{aligned}$$

□

Hull[1, sec. 10.4] shows the put-call parity for dividend-free shares and $b = (\mathfrak{n} = T)$, but without explicitly stating the preconditions or that the statement actually forms an equivalence.

Remark 4.25. For $\Delta t :: \texttt{TimeDiff}$ one receives a time-offset variant as

$$\begin{aligned} \mathrm{C}_{\Delta t,K,x} &:= \mathfrak{n} \rightsquigarrow \lambda t.\ \mathrm{C}_{(\mathfrak{n}=t+\Delta t),K,x} \\ \mathrm{P}_{\Delta t,K,x} &:= \mathfrak{n} \rightsquigarrow \lambda t.\ \mathrm{P}_{(\mathfrak{n}=t+\Delta t),K,x} \end{aligned}$$

and if $\texttt{after}\ \Delta t\ x \approx x$, then one receives

$$\mathrm{P}_{\Delta t,K,x} + x \ \approx\ \mathrm{C}_{\Delta t,K,x} + K^b.$$

This is easy to see by showing it under conditions $(\mathfrak{n} = t)$ for all $t :: \texttt{Time}$ using the previous theorem and then applying the quantification corollary 3.37.

4.6 American options, Merton's theorem

The options in the previous section being called "European" suggests that there is another type of options, namely *American options* which grant the holder the additional right to exercise before maturity:

Definition 4.26. A *American call option* is a contract that grants the holder the right, but not the obligation, to buy a certain asset until and including a certain point in the future for a fixed price K. A *American put option* grants the holder to sell the asset for a fixed price.

Define the following helper function (general American option):

$$\begin{aligned} \texttt{american} &:: \texttt{OB} \to \texttt{Con} \to \texttt{Con} \\ \texttt{american}\ b\ y &= \mathfrak{A}\ y_b \\ &\quad \text{where } y_b = \texttt{cond}\ (\bar{\mathfrak{e}}\ b)\ 0\ y \end{aligned}$$

Now define:

$$\begin{aligned} \texttt{americanCall} &:: \texttt{OB} \to \mathbb{R}^+ \to \texttt{Con} \to \texttt{Con} \\ \texttt{americanCall}\ b\ K\ x &= \texttt{american}\ b\ ((x-K) \vee 0) \end{aligned}$$

$$\begin{aligned} \texttt{americanPut} &:: \texttt{OB} \to \mathbb{R}^+ \to \texttt{Con} \to \texttt{Con} \\ \texttt{americanPut}\ b\ K\ x &= \texttt{american}\ b\ ((K-x) \vee 0) \end{aligned}$$

Also write $\mathrm{C}^{b,K,x}$ for $\texttt{americanCall}\ b\ K\ x$ and $\mathrm{P}^{b,K,x}$ for $\texttt{americanPut}\ b\ K\ x$.

Due to the nature of `anytime` options, we could have left out the "$\vee 0$" parts here, but they will prove useful for conformity with European options.

So $\texttt{american}\ b\ y$ gives the holder the option to acquire y up to and including the first occurrence of b. As usual, in the literature, $b = (\mathfrak{n} = T)$ for some $T :: \texttt{Time}$, but my approach allows more general conditions.

The aim of this section is now to prove the following statement, which is sometimes attributed to Merton:

> Given nonnegative interest rates, an American call option on a dividend-free share is never exercised before maturity. Hence, the American and European call options are equal in present value.

I start with a few preparations:

First, for working with `american`, one needs to restrict b to "reasonable" conditions. As an example, consider $b = (\mathfrak{n} > T)$: $\texttt{american}\ b\ y$ is thought to be valid until and including the *first* occurrence of b. However, it is not clear what this is supposed to mean unless time is discrete. For these "strange" b, Merton's theorem does not hold and one needs to exclude them as follows:

Definition 4.27. An observable $b :: \texttt{OB}$ is called *initialized* if

$$\mathfrak{e}\ b = \mathfrak{e}\ (\mathfrak{f}\ b).$$

Lemma 4.28. *Let* $y :: \texttt{Con}$ *and* $b :: \texttt{OB}$ *initialized. Then*

$$\mathfrak{W}'\ b\ y_b \approx_{\neg\bar{\mathfrak{e}}\ b} \mathfrak{W}'\ b\ y.$$

Proof. As b is initialized, the statement is equivalent to

$$\mathfrak{W}'\ (\mathfrak{f}\ b)\ y_b \approx_{\neg\bar{\mathfrak{e}}\ (\mathfrak{f}\ b)} \mathfrak{W}'\ (\mathfrak{f}\ b)\ y.$$

To see this for the condition, note that $\bar{\mathfrak{e}}\ b = \bar{\mathfrak{e}}\ (\mathfrak{e}\ b) = \bar{\mathfrak{e}}\ (\mathfrak{e}\ (\mathfrak{f}\ b)) = \bar{\mathfrak{e}}\ (\mathfrak{f}\ b)$.

By lemma 3.21, it now suffices to show

$$y_b \approx_{\mathfrak{f}\ b} y,$$

which is clear by definition of y_b since $\mathfrak{f}\ b \Rightarrow \neg\bar{\mathfrak{e}}\ b$. □

Remark 4.29. Lemma 4.28 is wrong if b is not initialized:

For example, consider a model where time is dense after t, i.e.

$$\forall t' > t \, \exists t'' : t < t'' < t'$$

and let $b = \mathfrak{n} > t$. Then it is easy to see that $\bar{\mathfrak{e}}\ b = \mathfrak{n} > t = b$ and hence $\neg\bar{\mathfrak{e}}\ b = \mathfrak{n} \leq t = \neg b$. Note also that $b = \mathfrak{e}\ b$.

We have $y_b = \texttt{cond}\ b\ 0\ y \approx_b 0$ and so $\mathfrak{W}'\ b\ y_b \approx \mathfrak{W}'\ b\ 0 \approx 0$ by lemma 3.21. But the RHS $\mathfrak{W}'\ b\ y$ is not in general 0 which can be seen from the properties of the `next` operation in section 3.4.

Lemma 4.30. *Let $y \succeq 0$ and $b, c :: \texttt{OB}$ be arbitrary. Then*

1. $\mathfrak{W}'\ c\ y_b \preceq_{\mathfrak{e}\ b} y_b$.
2. $\mathfrak{W}'\ c\ (\mathfrak{W}'\ b\ y_b) \preceq \mathfrak{W}'\ b\ y_b$

Proof. 1:

Use case distinction for $\mathfrak{e}\ b = \bar{\mathfrak{e}}\ b \vee \mathfrak{f}\ b \Leftarrow \bar{\mathfrak{e}}\ b \vee \mathfrak{e}\ c \vee (\neg\mathfrak{e}\ c \wedge \mathfrak{f}\ b)$.

For $\mathfrak{e}\ c$, the statement is trivial.

For $\bar{\mathfrak{e}}\ b$, note that $y_b \approx_{\bar{\mathfrak{e}}\ b} 0$ by definition and $\bar{\mathfrak{e}}\ b = \mathfrak{e}\ (\bar{\mathfrak{e}}\ b)$, so one can apply the monotonicity lemma 3.21 to receive

$$\mathfrak{W}'\ c\ y_b \approx_{\bar{\mathfrak{e}}\ b} \mathfrak{W}'\ c\ 0 \approx 0 \approx_{\bar{\mathfrak{e}}\ b} y_b.$$

For $f := \neg\mathfrak{e}\ c \wedge \mathfrak{f}\ b$, note the following:

1. $f \Rightarrow \neg\bar{\mathfrak{e}}\ b$ and $y_b \approx_{\neg\bar{\mathfrak{e}}\ b} y$ by definition. Further, $y \succeq 0$ by assumption.
2. $f \Rightarrow b \wedge \neg\mathfrak{e}\ c \Rightarrow \mathfrak{e}\ (b \wedge \neg\mathfrak{e}\ c)$ and

   $$\mathfrak{W}'\ c\ y_b \approx_{\mathfrak{e}\ (b \wedge \neg\mathfrak{e}\ c)} \mathfrak{W}'\ c\ 0 \approx 0.$$

   To see this, note that $\mathfrak{e}\ (b \wedge \neg\mathfrak{e}\ c) \wedge \mathfrak{e}\ c \Rightarrow \bar{\mathfrak{e}}\ b$ by lemma 2.25 and recap that $y_b \approx_{\bar{\mathfrak{e}}\ b} 0$ by definition. Then apply monotonicity of $\mathfrak{W}'\ c$ (lemma 3.21).

In total we have

$$y_b \approx_f y \succeq 0 \approx_f \mathfrak{W}'\ c\ y_b.$$

2:

By lemma 3.22, $\mathfrak{W}'\ c\ (\mathfrak{W}'\ b\ y_b) \approx \mathfrak{W}'\ b\ (\mathfrak{W}'\ c\ y_b)$. Now, applying monotonicity of $\mathfrak{W}'\ b$ to part 1, the claim follows. □

Remark 4.31. From the proof of the previous theorem, it is clear that the assumptions could be weakened to $y \succeq_{\neg\bar{\mathfrak{e}}\ b} 0$ or even $y \succeq_{\neg\mathfrak{e}\ c \wedge \mathfrak{f}\ b} 0$.

As expected, one receives the "easy direction" that the American option is always worth at least as much as the European one. Recap that the "general" version of the European options corresponding to `american` is simply $\mathfrak{W}'$.

Theorem 4.32. *Let $b :: \texttt{OB}$ be initialized.*

1. *If $y :: \texttt{Con}$, then*

   $$\texttt{american}\ b\ y \succeq_{\neg\bar{\mathfrak{e}}\ b} \mathfrak{W}'\ b\ y.$$

2. If $K :: \mathbb{R}^+$ *and* $x ::$ `Con`*, then*

$$\begin{aligned} \mathrm{C}^{b,K,x} &\succeq_{\neg \bar{\mathfrak{e}}\ b} \mathrm{C}_{b,K,x} \\ \mathrm{P}^{b,K,x} &\succeq_{\neg \bar{\mathfrak{e}}\ b} \mathrm{P}_{b,K,x}. \end{aligned}$$

Proof. 1:

$$\begin{aligned} \texttt{american}\ b\ y = \quad & \mathfrak{A}\ y_b \\ \succeq \quad & \mathfrak{W}'\ b\ (\mathfrak{A}\ y_b) \succeq \mathfrak{W}'\ b\ y_b \\ \approx_{\neg \bar{\mathfrak{e}}\ b} & \mathfrak{W}'\ b\ y \end{aligned}$$

by lemma 4.28.

2 now follows directly from the definitions of the involved contracts. □

Remark 4.33. The $\neg \bar{\mathfrak{e}}\ b$ constraint above is due to a technical detail: An American option acquired strictly after maturity, i.e. in a situation where $\bar{\mathfrak{e}}\ b$ holds, is worthless, while for a European option as defined above, the underlying contract is acquired immediately.

One receives a variant of `american` which mirrors the above behavior of the European option by

$$\texttt{american}'\ b\ y := \texttt{cond}\ (\mathfrak{e}\ b)\ (y \vee 0)\ (\texttt{american}\ b\ y).$$

For this variant, one receives "$\succeq$" without side conditions in theorem 4.32. I use `american` here for simplicity.

Theorem 4.34 (Merton, general version)**.** *Let* $b ::$ `OB` *be initialized and* $y ::$ `Con` *be such that* $0 \preceq y \preceq \mathfrak{W}'\ b\ y$*. Then*

$$\texttt{american}\ b\ y \preceq \mathfrak{W}'\ b\ y.$$

As $\mathfrak{W}'\ b\ y \preceq_{\neg \bar{\mathfrak{e}}\ b} \texttt{american}\ b\ y$ by theorem 4.32, we receive equality in present value under conditions $\neg \bar{\mathfrak{e}}\ b$.

Proof. Perform case distinction on $\top = \bar{\mathfrak{e}}\ b \vee \neg \bar{\mathfrak{e}}\ b$:

For $\bar{\mathfrak{e}}\ b$, we have $y \succeq 0$ and hence also $\mathfrak{W}'\ b\ y \succeq \mathfrak{W}'\ b\ 0 \approx 0$. On the other hand, $y_b \approx_{\bar{\mathfrak{e}}\ b} 0$ and hence also $\texttt{american}\ b\ y = \mathfrak{A}\ y_b \approx_{\bar{\mathfrak{e}}\ b} 0$ (via monotonicity. We have $\bar{\mathfrak{e}}\ b = \mathfrak{e}\ (\bar{\mathfrak{e}}\ b)$.)

So consider $\neg \bar{\mathfrak{e}}\ b$. By lemma 4.28 and definition of `american` it suffices to show that

$$\mathfrak{A}\ y_b \preceq \mathfrak{W}'\ b\ y_b.$$

To that end, I use minimality of `anytime` (axiom (*3.31)). One has to show the following:

1. $\mathfrak{W}'\ b\ y_b \succeq y_b$
2. $\mathfrak{W}'\ b\ y_b \succeq \mathfrak{W}'\ c\ (\mathfrak{W}'\ b\ y_b)$ for all $c ::$ `OB`.

2 is just lemma 4.30.2.

1 follows from the assumption as follows: Do case distinction on $\top = \mathfrak{e}\ b \vee \neg\bar{\mathfrak{e}}\ b$. For $\mathfrak{e}\ b$, the statement is trivial, so consider $\neg\bar{\mathfrak{e}}\ b$.

Since we have $y \succeq 0$, also $y \succeq y_b$ (cf. definition of y_b) and so

$$y_b \preceq y \preceq \mathfrak{W}'\ b\ y \approx_{\neg\bar{\mathfrak{e}}\ b} \mathfrak{W}'\ b\ y_b$$

where the second relation is by assumption and the third is lemma 4.28. □

Corollary 4.35 (Merton)**.** *Assume non-negative interest rates.*

Let $b :: \texttt{Con}$ *be initialized and let* $x :: \texttt{Con}$ *be such that* $\mathfrak{W}'\ b\ x \succeq x$*. Let* $K :: \mathbb{R}^+$*. Then*

$$\mathrm{C}^{b,K,x} \preceq \mathrm{C}_{b,K,x}.$$

As for the general case, we receive equality in present value under conditions $\neg\bar{\mathfrak{e}}\ b$ together with theorem 4.32.

Proof. Non-negative interest rates imply that $\mathfrak{W}'\ b\ \texttt{one} \preceq \texttt{one}$. Hence

$$x - K \preceq \mathfrak{W}'\ b\ x - \mathfrak{W}'\ b\ K \approx \mathfrak{W}'\ b\ (x - K).$$

By $0 \approx \mathfrak{W}'\ b\ 0$ one then receives

$$0 \vee (x - K) \preceq \mathfrak{W}'\ b\ 0 \vee \mathfrak{W}'\ b\ (x - K) \preceq \mathfrak{W}'\ b\ (0 \vee (x - K))$$

where the last relation is lemma 3.24.

Hence, $0 \vee (x - K)$ is suitable for theorem 4.34 and the result follows directly from the definition of $\mathrm{C}^{b,K,x}$ and $\mathrm{C}_{b,K,x}$. □

As for forward prices and the put-call parity, Hull[1, sec. 10.5] states the above corollary for dividend-free shares and $b = (\mathfrak{n} = T)$.

Remark 4.36. For $\Delta t :: \texttt{TimeDiff}$ one receives a time-offset variant like for European options as

$$\mathrm{C}^{\Delta t,K,x} := \mathfrak{n} \rightsquigarrow \lambda\, t.\ \mathrm{C}^{(\mathfrak{n}=t+\Delta t),K,x}.$$

It is clear that $(\mathfrak{n} = t + \Delta t)$ is initialized and if $\texttt{after}\ \Delta t\ x \succeq x$, then one again receives

$$\mathrm{C}^{\Delta t,K,x} \preceq \mathrm{C}_{\Delta t,K,x}.$$

One can also consider the *next* occurrence of an event $b :: \texttt{OB}$ via

$$\widetilde{\mathrm{C}}_{b,K,x} := \mathfrak{n} \rightsquigarrow \lambda\, t.\ \mathrm{C}_{(b \wedge \mathfrak{n} \geq t),K,x}$$
$$\widetilde{\mathrm{C}}^{b,K,x} := \mathfrak{n} \rightsquigarrow \lambda\, t.\ \mathrm{C}^{(b \wedge \mathfrak{n} \geq t),K,x}.$$

If b is now such that $b \wedge \mathfrak{n} \geq t$ is initialized for any $t :: \texttt{Time}$, $\mathfrak{W}\ b\ x \succeq x$ and non-negative interest rates are assumed, then

$$\widetilde{\mathrm{C}}^{b,K,x} \preceq \widetilde{\mathrm{C}}_{b,K,x}.$$

One receives the converse of the two variants of Merton's theorem as well if one can assume the involved contracts to be non-negative in present value:

Theorem 4.37. *Let* $b :: \texttt{OB}$ *be initialized.*

1. *If* $y :: \texttt{Con}$ *is such that*

$$\texttt{american}\ b\ y \preceq \mathfrak{W}'\ b\ y,$$

then $y \preceq \mathfrak{W}'\ b\ y$.

2. *If* $x :: \texttt{Con}$ *is such that* $x \succeq 0$ *and*

$$\mathrm{C}^{b,K,x} \preceq \mathrm{C}_{b,K,x}$$

for any $K :: \mathbb{R}^+$*, then* $x \preceq \mathfrak{W}'\ b\ x$.

Proof. 1: Case distinction on $\top = \mathfrak{e}\ b \vee \neg\bar{\mathfrak{e}}\ b$. Under $\mathfrak{e}\ b$, the statement is trivial. Under $\neg\bar{\mathfrak{e}}\ b$ we have

$$y \approx_{\neg\bar{\mathfrak{e}}\ b} y_b \preceq \mathfrak{A}\ y_b = \texttt{american}\ b\ y \preceq \mathfrak{W}'\ b\ y.$$

2: The assumption for $K = 0$ means that part 1 applies to $0 \vee (x - 0) \approx x$. □

4.7 A definition for dividend-free shares

Hull [1] states that the put-call parity and Merton's theorem should hold whenever x is a dividend-free share and $b = (\mathfrak{n} = t)$ for some $t :: \texttt{Time}$. From this assumption, one can vice versa characterize what it should actually mean for x to be a dividend-free share:

- From theorem 4.20, we know that the theorem on forward prices is equivalent to $x \approx \mathfrak{W}'\ (\mathfrak{n} = t)\ x$.
- From theorem 4.24, we know that the PCP is also equivalent to $x \approx \mathfrak{W}'\ (\mathfrak{n} = t)\ x$.
- Then Merton's theorem 4.35 follows as well.

The case where x is only known to be dividend-free until a certain point in time is covered here as well: Then the statements only hold certain t.

So the following seems to be a reasonable definition:

Definition 4.38. A contract x is called a *dividend-free share* if $x \approx \mathfrak{W}'\ (\mathfrak{n} = t)\ x$ for any t.

x is called *dividend-free until* $T :: \texttt{Time}$ if the above holds for any $t \leq T$.

5 A probabilistic model for LPT

The aim of this section is to show that a generalized version of the binomial model, i.e. a probabilistic model with finite states and finite time, is in fact a model for LPT.

Peyton Jones and Eber [3] gave a sketch of the binomial model for their framework and claimed that any other model could be simply "plugged in". In the light of theorem 5.1 below, this claim seems doubtable: As soon as sample spaces become uncountable, it is not clear what a sensible definition of the `Obs` monad could look like.

Let $\mathcal{M}$ be the category of measurable spaces and maps where a measurable space $X \in \mathcal{M}$ is a pair $(X, \mathcal{A}(X))$ where $\mathcal{A}(X)$ is a σ-algebra on X. In the following section, I will develop a model $\mathscr{A}$ of LPT with the following properties:

- $\mathscr{A}$ is based in the category $\mathcal{M}$, i.e. the interpretation of any sort is a measurable space and the interpretation of any functional term is a measurable map.
- Every contract has a price, i.e. all the contracts are, up to present value, of form `moneyG` o for some o :: `Obs` $\mathbb{R}$.
- Observables are stochastic processes on their respective value types.
- Contracts are stochastic processes on $\mathbb{R}$ which define their present values.

The construction follows the three steps in the construction of LPT: First I show how the primitive types and ADTs can be modeled in $\mathcal{M}$ ($\mathrm{LPT}_{\mathrm{Prim}}$). Then I define the monad $\mathcal{RV}$ of random variables on a certain class of sample spaces and – based on that – the monad $\mathcal{PR}$ of stochastic processes which will model the `Obs` type ($\mathrm{LPT}_{\mathrm{Obs}}$). Finally, I show how to model contracts.

The presented class of models for the theory of observables will be more general in that it allows infinite states and infinite time. In the model for the full theory with contracts, everything will be finite.

5.1 The primitive types as measurable spaces

For the numeric types, one can simply choose the obvious models which their natural σ-algebras: The set $\mathbb{R}$ of real numbers for the numeric type $\mathbb{R}$, $\mathbb{R}^+$ for $\mathbb{R}^+$, $\mathbb{R} \setminus \{0\}$ for $\mathbb{R}^*$ etc.

For the `Time` type, different options like $\mathbb{Z}$, $\mathbb{Q}$, and $\{1, \ldots, T\}$ (with their natural σ-algebras) will be discussed below. For `TimeDiff`, one can choose e.g. $\mathbb{Z}$.[29]

The theory presented above uses two algebraic data types, `Bool` and `Maybe` a (`Maybe` a is only used in `timeOffset`). Models in $\mathcal{M}$ for these are easily constructed as follows:

`Bool`$^{\mathscr{A}}$ should be the discrete two-point space, of course.

[29] Recap that `Time` and `TimeDiff` are connected only by the `timeOffset` map which is of `Maybe` result type, so one is free to choose `TimeDiff` whatever fits best.

$(\texttt{Maybe}\ a)^{\mathscr{A}}$**,** where a is some sort the interpretation $a^{\mathscr{A}}$ of which is already defined as a measurable space, should be the disjoint union $a^{\mathscr{A}} \dot\cup \{*\}$ where $*$ is a distinguished point not in $a^{\mathscr{A}}$.

Then define $\texttt{Nothing}_a{}^{\mathscr{A}} := *$ and let $\texttt{Just}_a{}^{\mathscr{A}} : a^{\mathscr{A}} \to a^{\mathscr{A}} \dot\cup \{*\}$ be the inclusion. Finally, define the $\texttt{case}_{\texttt{Maybe},a}$ schema by the universal property of the coproduct "$\dot\cup$" in $\mathcal{M}$.

Sensible measurable spaces for general algebraic data types can be defined by first translating a ADT definition into a much more general combinatorial framework called *species*, then providing measurable spaces for these. The translation mechanism exposes a rich structure. It can be found in appendix D.

5.2 Observables as stochastic processes

In the following, I want to define a monad $\mathcal{RV}$ on $\mathcal{M}$ such that $\mathcal{RV}\ X$ is the set of random variables, i.e. measurable maps from a certain sample space Ω to X, together with a suitable σ-algebra. In a second step, I will then take products of these monads with respect to a filtration to receive the monad $\mathcal{PR}$ of stochastic processes which will model `Obs`.

However, such a σ-algebra does not exist for every Ω. Aumann [10] showed the following:

Theorem 5.1 (Aumann)**.** *Let J be the two-element space and I the unit interval with their natural σ-algebras. Let J^I be the set of measurable functions $I \to J$. Then there is no σ-algebra on J^I such that the* evaluation map

$$\begin{aligned} \varepsilon : I \times J^I &\to J \\ \varepsilon(x, f) &:= f(x) \end{aligned}$$

is measurable.

Aumann's paper provides a detailed discussion of which subsets of maps allow evaluation.

As the $\mathcal{RV}$ monad should describe the space of *all* random variables and I require evaluation below to define the `join` operation, one needs to make a restriction on the sample space:

Definition 5.2. Let $\Omega \in \mathcal{M}$ be a measurable space. An element $A \in \mathcal{A}(\Omega)$ is called an *atom* if $A \neq \emptyset$ and the only measurable proper subset of A is $\emptyset$.

Ω is called *atomic* if any element of Ω is contained in an atom. Ω is called an *admissible sample space* if it is atomic with at most countably many atoms.

It is clear that atoms are pairwise disjoint. All spaces considered in this section are atomic. The below theorem 5.13 will show that admissible sample spaces allow evaluation.

Remark 5.3. An admissible sample space Ω with $\mathbb{K} \in \mathbb{N} \cup \{\mathbb{N}\}$ atoms is isomorphic up to indistinguishability to $(\mathbb{K}, \mathcal{P}(\mathbb{K}))$. Here, two maps $f, g :: X \to Y$ are called *indistinguishable* if for all $x \in X$ and $A \in \mathcal{A}(Y)$ we have $f(x) \in A \Leftrightarrow g(x) \in A$.

To see this, fix an enumeration $\{K_i \mid i \in \mathbb{K}\}$ of the atoms of Ω and choose for any i some $\omega_i \in K_i$. Then define

$$\begin{aligned} f &: \Omega \to \mathbb{K} \\ f(\omega) &:= i \text{ if } \omega \in K_i \\ \\ g &: \mathbb{K} \to \Omega \\ f(i) &:= \omega_i \end{aligned}$$

These maps are measurable: f is measurable because any preimage of a subset of $\mathbb{K}$ is a (countable) union of atoms (which are measurable sets). g is measurable because $\mathbb{K}$ is discrete.

Clearly $f \circ g = \mathtt{id}_{\mathbb{K}}$. $g \circ f$ is indistinguishable from $\mathtt{id}_\Omega$: $(g \circ f)(\omega)$ is in the same atom as ω, hence these two points cannot be distinguished by a measurable set (easy to see / cf. below).

Note that the maps f and g are not canonical, which is the reason one cannot just replace any admissible sample space with its $\mathbb{K}$ space. For example, if there are two admissible σ-algebras $\mathcal{A} \subseteq \mathcal{B}$ on Ω, the correspondence would not reflect the relationship between $\mathcal{A}$ and $\mathcal{B}$. However, it is a helpful piece of intuition that admissible sample spaces behave "essentially like discrete countable spaces". In particular, any random variable $h : \Omega \to X$ must factor over $\mathbb{K}$ up to indistinguishability. If X has atoms as points such as $\mathbb{R}$, this factorization must be exact, so h is essentially a map on a countable set.

Note that $\mathbb{R}$ itself is not admissible.

Fix a measurable space Ω admissible as of above. Usually, one also would fix a probability measure, but these are not important yet.

5.2.1 A few notes on atomic measurable spaces

To prepare for the following arguments, I give some general lemmas about atomic measurable spaces. All lemmas below are standard. The longer proofs can be found in appendix C.

Lemma 5.4. *Let $X_1, \ldots, X_n, Y$ be measurable spaces and let $f : X_1 \times \ldots \times X_n \to Y$ be a map of sets defined by application of certain (fixed) measurable functions to n variables. Then f is measurable.*

The lemma states that measurable functions can be combined to terms. Note how this property is crucial to arrive at a model where any term is measurable: It suffices to give measurable functions for all the functional *symbols* defined in LPT.

Proof. The lemma is *almost* trivial in that such a function is *almost* a chain of measurable functions. A hidden point is that a variable may be used more than once, for example in $f := (x \mapsto g(x, h(x, x)))$. One can arrive at a true chain of measurables by using the function

$$\begin{aligned} \Delta : &\quad X \to X \times X \\ \Delta(x) &:= (x, x). \end{aligned}$$

Now the above example is $f = g \circ (\mathtt{id}, h) \circ (\mathtt{id}, \Delta) \circ \Delta$.

Δ is measurable for any space X because the generators of $X \times X$ are rectangles of form $A \times B$, A, B measurable, and $\Delta^{-1}(A \times B) = A \cap B$ is measurable. □

Lemma 5.5. *Let $X = (X, \mathcal{A})$ be an atomic measurable space.*

Two elements $x, y \in X$ are called $\mathcal{A}$-indistinguishable *if there is no $A \in \mathcal{A}$ such that $x \in A$ and $y \notin A$.*

1. *For any $x \in X$ there is a unique atom $K_x := K_x^{\mathcal{A}} \in \mathcal{A}$ containing x.*
2. *Any measurable set is a union of atoms.*
3. *For $x, y \in X$, x, y are indistinguishable iff they lie in the same atom iff $K_x = K_y$.*
4. *If $f : X \to Y$ is a measurable map and $x, y \in X$ are indistinguishable, then $f(x), f(y)$ are indistinguishable. If $K \in \mathcal{A}$ is an atom, then $f[K]$ does not have proper, nonempty subsets in $\mathcal{B}$.*

Note that $f[K]$ is in general not element of $\mathcal{A}(Y)$, hence can't be called "atom".

Definition 5.6. If X and Y are spaces, $E \subseteq X \times Y$ and $A \subseteq X$, define the *section*

$$E_A := \{y \in Y \mid A \times \{y\} \subseteq E\}.$$

Define E_A for $A \subseteq Y$ analogously. If $x \in X$, write short $E_x := E_{\{x\}}$.

Lemma 5.7. *Let $X, Y \in \mathcal{M}$ be atomic. Then the following holds for the product space $X \times Y$:*

1. *The atoms of $X \times Y$ are of form $K_X \times K_Y$ where $K_X \subseteq X$ and $K_Y \subseteq Y$ are atoms. In particular, $X \times Y$ is atomic.*
2. *Let $E \subseteq X \times Y$ be measurable and $K \subseteq X$ be an atom. Then E_K is measurable. If $x \in X$, then $E_x = E_{K_x}$ is measurable. (analogously for Y)*

Corollary 5.8 (Partial Application in $\mathcal{M}$)**.** *If $f : X \times Y \to Z$ is a measurable function and $x \in X$, then*

$$\begin{aligned} f(x, \cdot) : \quad & Y \to Z \\ f(x, \cdot)(y) := \; & f(x, y) \end{aligned}$$

is measurable.

Remark 5.9. The converse of corollary 5.8 is wrong in general, i.e. if $f : X \times Y \to Z$ is a map of sets such that for any x, $f(x, \cdot)$ is measurable, this does not in general make f measurable.

To see this, let $X = Y$ be a space of cardinality strictly greater than the cardinality of the continuum $|\mathbb{R}|$ such that points are measurable in X. Let Z be the two-element space `Bool` and define

$$f(x, y) := \begin{cases} \texttt{True} & \text{if } x = y \\ \texttt{False} & \text{otherwise} \end{cases}.$$

Then for any x, $f(x, \cdot)(y) = \texttt{True}$ iff $y = x$, so $f(x, \cdot)^{-1}(\{\texttt{True}\}) = \{x\}$, so $f(x, \cdot)$ is always measurable.

However,

$$f^{-1}(\{\texttt{True}\}) = \{(x, x) \mid x \in X\}$$

is *not* measurable. This is called *Nedoma's Pathology*.[30]

The above remark is the main reason why general higher-order functions are problematic in $\mathcal{M}$. For example, if ζ assigns to a function $g : Y \to Z$ a function $\zeta\ g : Y' \to Z'$ in some way, then g could be of form $f(x, \cdot)$ for f as above. Then the functions $\zeta\ f(x, \cdot)$ are all measurable, but the function

$$(x, y) \mapsto (\zeta\ f(x, \cdot))(y)$$

is *not* in general measurable. So these are unexpectedly hard to combine.

As a reaction, I introduced the closure emulation schema (cf. section A.1.3) which will be mentioned again in section 5.2.2 below.

Corollary 5.10. *Let X be atomic, Ω an admissible sample space and $E \subseteq X \times \Omega$. Let $A \subseteq X$ and $B \subseteq \Omega$ be measurable. Then the following sets are measurable:*

1. $E_B = \{x \in X \mid \{x\} \times B \subseteq E\}$
2. $E_A = \{\omega \in \Omega \mid A \times \{\omega\} \subseteq E\}$
3. $\pi_1[E] = \{x \in X \mid \exists \omega \in \Omega : (x, \omega) \in E\}$
4. $\pi_2[E] = \{\omega \in \Omega \mid \exists x \in X : (x, \omega) \in E\}$

Note that corollary 5.10 is *wrong* in general if Ω is not admissible. E.g. application of 3 to $\mathbb{R} \times \mathbb{R}$ would mean that every analytic subset of $\mathbb{R}$ is Borel, which is famously not true and led to the development of descriptive set theory.[31]

5.2.2 The monad of random variables

Notation 5.11. From now on, for the sake of readability, I will leave out the interpretation marker $\cdot^{\mathscr{A}}$ in some cases. E.g. I will write just `return` instead of $\texttt{return}^{\mathscr{A}}$.

With the preparation in place, one can define the monad of random variables:

Definition 5.12. Given $X \in \mathcal{M}$, define $\mathcal{RV}^{\Omega}\ X$ to be the following space:

$\mathcal{RV}^{\Omega}\ X$ is the set of measurable functions $p : \Omega \to X$, i.e. the set of morphisms $\mathrm{Hom}\,(\Omega, X)$ in $\mathcal{M}$.

$\mathcal{A}(\mathcal{RV}^{\Omega}\ X)$ is the σ-algebra generated by the sets $B_{\omega,A} := \{p \in \mathcal{RV}^{\Omega}\ X \mid p(\omega) \in A\}$ for $\omega \in \Omega$ and $A \in \mathcal{A}(X)$. This is equivalent to saying that $\mathcal{A}(\mathcal{RV}^{\Omega}\ X)$ is generated by the maps $(p \mapsto p(\omega))$ for $\omega \in \Omega$.

[30] Nedoma [11] showed this in 1957. Schechter [12, p. 550] gives the proof in a somewhat more accessible form.

[31] Cf. [13, thm. 14.2] for the statement and [13] in general for the topic of descriptive set theory.

When the space Ω is clear from the context, also write $\mathcal{RV}$ for $\mathcal{RV}^\Omega$.

If $f : X \to Y$ is measurable, let

$$\begin{aligned} \mathcal{RV}\ f &:= \texttt{fmap}\ f : \quad \mathcal{RV}\ X \to \mathcal{RV}\ Y \\ \texttt{fmap}\ f\ o &:= f \circ o \end{aligned}$$

For $X \in \mathcal{M}$ let

$$\begin{aligned} \texttt{return}_X : &\quad X \to \mathcal{RV}\ X \\ \texttt{return}_X\ x &:= (\omega \mapsto x) \\ \\ \texttt{join}_X : &\quad \mathcal{RV}\ (\mathcal{RV}\ X) \to \mathcal{RV}\ X \\ \texttt{join}_X\ p &:= (\omega \mapsto p(\omega)(\omega)) \end{aligned}$$

and write `return` and `join` without the subscripts when the context is clear.

It is easy to see that `fmap`, `return` and `join` fulfill the functor laws (*Fu1) and (*Fu2), the monad laws (*Mo1)–(*Mo5) and the laws (*Ob1)–(*Ob3) from section 2. Showing that the functions are well-defined and in category $\mathcal{M}$ again, i.e. measurable, requires some work though.

I first show two core statements about the structure of $\mathcal{RV}\ X$ all the other statements will reduce to.

Theorem 5.13. *Let $(X, \mathcal{A})$ be a measurable space. Then the evaluation function*

$$\begin{aligned} \varepsilon : \Omega \times \mathcal{RV}\ X &\to X \\ \varepsilon(\omega, o) &:= o(\omega) \end{aligned}$$

is measurable.

Proof. If $A \in \mathcal{A}$ and $K \in \mathcal{A}(\Omega)$ is an atom, then

$$B_{K,A} := \{p \in \mathcal{RV}\ X \mid p\,[K] \subseteq A\}$$

is measurable:[32] If $K = K_\omega$, then by lemma 5.5 $B_{K,A} = B_{\omega,A}$.

I now show that

$$\varepsilon^{-1}(A) = \bigcup_{K \in \mathcal{A}(\Omega)\ \text{atom}} K \times B_{K,A}.$$

The RHS is a countable union of sets measurable in $\Omega \times \mathcal{RV}\ X$, hence measurable.

"$\supseteq$": If $(\omega, o) \in K \times B_{K,A}$, then by definition $\varepsilon(\omega, o) = o(\omega) \in A$.

"$\subseteq$": If $\varepsilon(\omega, o) = o(\omega) \in A$, then $o \in B_{\omega,A} = B_{K_\omega,A}$. Also, $\omega \in K_\omega$, so $(\omega, o) \in K_\omega \times B_{K_\omega,A}$. □

Lemma 5.14. *Fix an admissible sample space Ω and let $X, Y \in \mathcal{M}$ be atomic.*

1. *If $f : X \to \mathcal{RV}\ Y$ is measurable, then*

$$\begin{aligned} \texttt{uncurry}\ f : &\quad X \times \Omega \to Y \\ (\texttt{uncurry}\ f)(y, \omega) &:= f(y)(\omega) \end{aligned}$$

is measurable.

[32] In fact, it is easily seen using countability that all the sets $B_{B,A}$ for $B \in \mathcal{A}(\Omega)$ are measurable.

2. If $f : X \times \Omega \to Y$ is measurable, then

$$\begin{aligned} \texttt{curry}\ f : &\quad X \to \mathcal{RV}\ Y \\ (\texttt{curry}\ f)(x) &:= (\omega \mapsto f(x,\omega)) \end{aligned}$$

is well-defined and measurable.

As `curry` and `uncurry` are inverses to each other, this establishes a 1:1 correspondence between the two kinds of measurable maps.

Proof. 1: `uncurry` f can be described as a chain of measurable maps:

$$\begin{array}{ccc} X & \xrightarrow{f} & \mathcal{RV}\ Y \\ \times & & \times \xrightarrow{\varepsilon} Y \\ \Omega & \xrightarrow{\mathrm{id}} & \Omega \\ & \xrightarrow{\qquad \texttt{uncurry}\ f \qquad} & \end{array}$$

2: Well-definedness: Let $x \in X$. By corollary 5.8, then $(\texttt{curry}\ f)(x) = f(x, \cdot) : \Omega \to Y$ is measurable, i.e. $(\texttt{curry}\ f)(x) \in \mathcal{RV}\ Y$.

Measurability of `curry` f: Let $\omega \in \Omega$ and $A \subseteq Y$ measurable. I check preimages of the sets $B_{\omega,A} \subseteq \mathcal{RV}\ Y$. Let $x \in X$. We have

$$\begin{aligned} x \in (\texttt{curry}\ f)^{-1}(B_{\omega,A}) &\Leftrightarrow (\texttt{curry}\ f)(x)(\omega) \in A \\ &\Leftrightarrow f(x,\omega) \in A \\ &\Leftrightarrow x \in f^{-1}(A)_\omega \end{aligned}$$

which is measurable in X by lemma 5.7.2. □

Remark 5.15. The operations ε and `curry` above make $\mathcal{RV}\ X$ the *exponential object* X^Ω in the category $\mathcal{M}$. Recap that $\mathcal{M}$ does not have general exponential objects by Aumann's theorem 5.1, i.e. it is not *cartesian closed*.[33]

This is the main reason why one has to be "careful" when passing from a Haskell-style framework to category $\mathcal{M}$.

Note that the construction of `curry` f leads to a measurable function even if Ω is not admissible (the proof did not use admissibility). On the other hand, if `uncurry` f is always measurable, then so is $\varepsilon =$ `uncurry id`.

Remark 5.16. Lemma 5.4 said that any map that is given by a term of measurable functions is measurable. The above theorem 5.13 introduced a limited way of higher-orderness in that if one has variables of form $\mathcal{RV}\ X$ and Ω, one may perform application in the term. And the `curry` construction from lemma 5.14 essentially said that one may even introduce new variables of form Ω and receive an element of $\mathcal{RV}\ Y$ for some Y.

Without admissibility, one would still be allowed to apply a parameter of form $\mathcal{RV}\ X$ to some *fixed* $\omega \in \Omega$ (by definition of the σ-algebra on $\mathcal{RV}\ X$), but that ω could not be given as an argument to the function.

Theorem 5.17. *The following functions are well-defined and measurable:*

1. `fmap` $f : \mathcal{RV}\ X \to \mathcal{RV}\ Y$ *if* $f : X \to Y$ *is a measurable function*

[33] For the category-theoretic concepts, as usual, cf. [7].

2. $\texttt{return} : X \to \mathcal{RV}\ X$

3. $\texttt{join} : \mathcal{RV}\ (\mathcal{RV}\ X) \to \mathcal{RV}\ X$

$\mathcal{RV} : \mathcal{M} \to \mathcal{M}$ *is a monad.*

Proof. Via remark 5.16, these follow directly from the definitions of `fmap`, `return` and `join`.

When in doubt, note that it suffices to show that the `uncurry` variants are well-defined and measurable by lemma 5.14 and one has $\texttt{uncurry}\ (\texttt{fmap}\ f) = f \circ \varepsilon$, $\texttt{uncurry return} = \pi_1$ and $\texttt{uncurry join} = \varepsilon \circ (\varepsilon, \texttt{id}) \circ (\texttt{id}, \Delta)$ where $\Delta(\omega) := (\omega, \omega)$ and $\pi_1(x, \omega) := x$ □

Closure Emulation While now $\mathcal{RV}$ is a monad, my framework in fact requires something slightly stronger for `fmap`, namely that also the closure emulation schema from section A.1.3 is supported.

Consider a function $f : X \to Y$ where `fmap` is to be applied to. f may have arisen by partial application from $g : X \times Z \to Y$ as $f = g(\,\cdot\,, z)$ for some $z \in Z$ (cf. corollary 5.8). z could be called the *closure context* of g in `fmap`. The framework does not only require that $\texttt{fmap}\ f$ be measurable but even that it is created in a measurable way, i.e. that the map

$$\begin{aligned} \mathcal{RV}\ X \times Z &\to \mathcal{RV}\ Y \\ (o, z) &\mapsto \texttt{fmap}\ (g(\,\cdot\,, z))\ o \end{aligned}$$

be measurable so that a term like

$$\lambda\, z\ o.\ \texttt{fmap}\ (\lambda\, x.\ g\ x\ z)\ o$$

leads to a measurable map again. This is exactly what the following theorem 5.18 states.

As of section A.1.3, the framework in fact defines symbols for these functions (and not for `fmap`, which is higher order) as

$$\widetilde{\texttt{fmap}}_{\lambda\, x\ z.\ g\ x\ z}$$

and the above term is short for

$$\lambda\, z\ o.\ \widetilde{\texttt{fmap}}_{\lambda\, x\ z.\ g\ x\ z}\ z\ o.$$

Theorem 5.18 (Closure emulation for `fmap`)**.** *Let X, Y, Z be measurable spaces and let $g : X \times Z \to Y$ be measurable. Define*

$$\begin{aligned} \widetilde{\texttt{fmap}}_g :&\quad \mathcal{RV}\ X \times Z \to \mathcal{RV}\ Y \\ \widetilde{\texttt{fmap}}_g(o, z) &:= (\omega \mapsto g(o(\omega), z)) \\ &= \texttt{fmap}\ g(\,\cdot\,, z)\ o. \end{aligned}$$

Then $\widetilde{\texttt{fmap}}_g$ is measurable.

Proof. The proof goes exactly like in theorem 5.17. □

Again, the complication of `fmap` vs. $\widetilde{\texttt{fmap}}_g$ is introduced by the fact that $\mathcal{M}$ supports only *some* function spaces (exponential objects). For example, one can't state that "the map $(z \mapsto g(\cdot, z))$ is measurable" because there is in general no sensible σ-algebra on the set of measurable functions $X \to Y$.

Remark 5.19. One can show well-definedness and measurability of `fmap` f, `return` and $\widetilde{\texttt{fmap}}_g$ without using admissibility in an elementary way: We have $(\texttt{fmap}\ f)^{-1}(B_{\omega,A}) = B_{\omega,f^{-1}(A)}$, $\texttt{return}^{-1}(B_{\omega,A}) = A$ and $\widetilde{\texttt{fmap}}_g^{-1}(B_{\omega,A})$ can be shown to be measurable as well.

However, it is not clear whether `join` can be measurable while ε is not.

5.2.3 From random variables to stochastic processes

I will next extend the concept above to stochastic processes on a countable index set.

Let $\mathbb{T}$ be a totally ordered set of at most countable cardinality, e.g. $\{1, \ldots, T\}$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ or the ordinal number $\omega \cdot \omega$, and assume the discrete σ-algebra on $\mathbb{T}$.[34] Let $\mathbb{T}$ model the `Time` type. Let Ω be some set and fix a filtration $(\mathcal{F}_t)_{t\in\mathbb{T}}$ of Ω such that for any $t \in \mathbb{T}$, $(\Omega, \mathcal{F}_t)$ is an admissible sample space. I call $(\mathcal{F}_t)_{t\in\mathbb{T}}$ an *admissible filtration*.

Remark 5.20. One can see similarly to remark 5.3 that $(\Omega, (\mathcal{F}_t)_{t\in\mathbb{T}})$ is up to indistinguishability a tree with $\mathbb{T}$ levels and at most countably many branchings per level:

Consider the following partially ordered set:

- $V := \bigcup_{t\in\mathbb{T}}(\{t\} \times \{\text{atoms of } \mathcal{F}_t\})$
- $(t, K) \leq (s, L) :\Leftrightarrow t \leq s \wedge K \supseteq L$

Note that this defines indeed a partial order and the ordering is tree-like:[35] If $v_1, v_2, w \in V$ and $v_1, v_2 \leq w$, then $v_1 \leq v_2$ or $v_2 \leq v_1$. This follows from the fact that $(\mathcal{F}_t)_{t\in\mathbb{T}}$ is a filtration and V is defined on atoms. Note how each of the "levels" of V (sets with equal t) is countable and that there are $\mathbb{T}$ levels.

Let W be the set of maximal chains through V, i.e. the set of branches of V, such that the intersection of the second components of a chain in W is non-empty. Define sets

$$v\uparrow := \{w \in W \mid v \in w\}$$

for $v \in V$ and define a filtration $(\mathcal{G}_t)_{t\in\mathbb{T}}$ on W by letting $\mathcal{G}_t$ be generated by the sets $(t, K)\uparrow$ where K is an atom of $\mathcal{F}_t$. It is easy to see that these generators are atoms then, so $(W, \mathcal{G}_t)$ is admissible for all t.

Finally, define the following maps:

$$\begin{aligned} f &: \Omega \to W \\ f(\omega) &:= \{(t, K) \in V \mid \omega \in K\} \\ g &: W \to \Omega \\ g(w) &:= \omega \in \bigcap \{K \mid \exists t \in \mathbb{T} : (t, K) \in w\} \end{aligned}$$

[34] This is not the same as "discrete time", of course. $\mathbb{Q}$ can be used to receive – not continuous, but – dense time, i.e. there is never a "next" point in time.

[35] without a root unless $\mathbb{T}$ has a minimum which is assigned the trivial σ-algebra. A root is not required for the following argument.

The values of g leave a certain degree of freedom, so a (arbitrary) choice must be made here.

It is clear that f and g are well-defined. For any t, f is measurable with respect to $\mathcal{F}_t$ and $\mathcal{G}_t$ because $f^{-1}((t,K)\uparrow) = K$ and g is measurable because $g^{-1}(K) = (t,K)\uparrow$ if K is an atom of $\mathcal{F}_t$. We have that $f \circ g = \mathtt{id}_W$ and $f \circ g$ is indistinguishable from $\mathtt{id}_\Omega$ by any σ-algebra $\mathcal{F}_t$ because it maps atoms to themselves. Hence, f and g define a correspondence between the two filtrations as required.

So if the discrete space $\mathbb{N}$ is the canonical admissible sample space by remark 5.3, the tree $\mathbb{N}^{\mathbb{N}}$ is the canonical admissible filtration. Note that the limit, i.e. the σ-algebra induced by the union, of the filtration on the tree $\mathbb{N}^{\mathbb{N}}$ is the discrete space $\mathbb{N}^{\mathbb{N}}$, which is *not* admissible.

Using countability of $\mathbb{T}$, it is easy to see that the limit of $(W, (\mathcal{G}_t)_{t\in\mathbb{T}})$ is always discrete. Hence, the limit of $(\Omega, (\mathcal{F}_t)_{t\in\mathbb{T}})$ is admissible iff the corresponding W is countable.

One can now define the monad $\mathcal{PR}$ of random processes as the categorical *product* of the monads $\mathcal{RV}^{\mathcal{F}_t}$. This is known to exist for category $\mathcal{M}$.

Definition 5.21. Write just $\mathcal{RV}^{\mathcal{F}_t}$ for $\mathcal{RV}^{(\Omega,\mathcal{F}_t)}$.

Given $X \in \mathcal{M}$, define $\mathcal{PR}\ X$ to be the product space

$$\mathcal{PR}\ X := \mathcal{PR}^{(\Omega,(\mathcal{F}_t)_{t\in\mathbb{T}})}\ X := \bigtimes_{t\in\mathbb{T}} \mathcal{RV}^{\mathcal{F}_t}\ X.$$

The underlying set of this space is just the cartesian product of the spaces $\mathcal{RV}^{\mathcal{F}_t}\ X$. The σ-algebra $\mathcal{A}(\mathcal{PR}\ X)$ is the one generated by projections out of the product, i.e. generated by the sets

$$\begin{aligned} C_{t,A} &:= \{o \in \mathcal{PR}\ X \mid o_t \in A\} \\ &= \bigtimes_{t'\in\mathbb{T}} \begin{cases} A & \text{if } t' = t \\ \mathcal{RV}^{\mathcal{F}_{t'}}\ X & \text{otherwise} \end{cases} \end{aligned}$$

for $t \in \mathbb{T}$ and $A \in \mathcal{A}(\mathcal{RV}^{\mathcal{F}_t}\ X)$.

If $f : X \to Y$ is measurable, let

$$\begin{aligned} \mathcal{PR}\ f := \mathtt{fmap}\ f : &\quad \mathcal{PR}\ X \to \mathcal{PR}\ Y \\ (\mathtt{fmap}\ f\ o)_t := \mathtt{fmap}\ f\ o_t & \end{aligned}$$

For $X \in \mathcal{M}$ let

$$\begin{aligned} \mathtt{return}_X : &\quad X \to \mathcal{PR}\ X \\ (\mathtt{return}_X\ x)_t := \mathtt{return}_X\ x &\in \mathcal{RV}^{\mathcal{F}_t}\ X \end{aligned}$$

$$\begin{aligned} \mathtt{join}_X : &\quad \mathcal{RV}\ (\mathcal{RV}\ X) \to \mathcal{RV}\ X \\ (\mathtt{join}_X\ p)_t := \mathtt{join}_X\ p_t &\in \mathcal{RV}^{\mathcal{F}_t}\ X \end{aligned}$$

and write `return` and `join` without the subscripts again when the context is clear.

It is again clear as for $\mathcal{RV}$ (section 5.2.2) that $\mathcal{PR}$ fulfills the functor-, monad- and Ob laws. Well-definedness of the operations follows point-wise and measurability follows from the universal property of the product that measurable functions can be combined point-wise.

With the general monad operations set up, one can define the operation `ever` from section 2.3:

Lemma 5.22. *Define the following map:*

$$\begin{aligned} \texttt{ever}: &\quad \mathcal{PR}\ \texttt{Bool} \to \mathcal{PR}\ \texttt{Bool} \\ (\texttt{ever}\ b)_t(\omega) &:= \begin{cases} \texttt{True} & \text{if } \exists t' \leq t : b_{t'}(\omega) = \texttt{True} \\ \texttt{False} & \text{otherwise} \end{cases} \end{aligned}$$

`ever` *is a well-defined and measurable map.*

Proof. For $t \in \mathbb{T}$ consider

$$\begin{aligned} f_t: &\quad \mathcal{PR}^{(\mathcal{F}_{t'})_{t' \leq t}}\ \texttt{Bool} \to \mathcal{RV}^{\mathcal{F}_t}\ \texttt{Bool} \\ f_t(b) &:= \begin{cases} \texttt{True} & \text{if } \exists t' \leq t : b_{t'}(\omega) = \texttt{True} \\ \texttt{False} & \text{otherwise.} \end{cases} \end{aligned}$$

$f_t(b)$ is the lift "$\bigvee$" applied on top of b where

$$\begin{aligned} \bigvee: &\quad \texttt{Bool}^{\{t' \in \mathbb{T} | t' \leq t\}} \to \texttt{Bool} \\ \bigvee(\alpha) &:= \bigvee_{t' \leq t} \alpha_{t'} \end{aligned}$$

is measurable: $\bigvee^{-1}(\{\texttt{True}\}) = \bigcup_{t' \leq t} C_{t',\{\texttt{True}\}}$ which is a countable union by countability of $\mathbb{T}$.

So f_t is well-defined and measurable for any t. Now `ever` is just a combination of the maps

$$g_t: \quad \mathcal{PR}\ \texttt{Bool} \xrightarrow{\text{proj}} \mathcal{PR}^{(\mathcal{F}_{t'})_{t' \leq t}}\ \texttt{Bool} \xrightarrow{f_t} \mathcal{RV}^{\mathcal{F}_t}\ \texttt{Bool}$$

over $t \in \mathbb{T}$ via the product property. □

Finally, one can define `now` in the obvious way:

$$\begin{aligned} \texttt{now} &\in \mathcal{PR}\ \mathbb{T} \\ \texttt{now}_t &:= \texttt{return}\ t \in \mathcal{RV}^{\mathcal{F}_t}\ \mathbb{T} \end{aligned}$$

It is clear that this is well-defined.

It is easy to see that the above definitions of `ever` and `now` fulfill the axioms for the modal logic S4.3 in section 2.3 and the axioms for `now` in section 2.4: Simply fix a $\omega \in \Omega$ and consider the trajectories.

Altogether, one can model $\texttt{Obs}^{\mathscr{A}} = \mathcal{PR}$.

5.2.4 More about maps on $\mathcal{RV}\ X$

Using the results from the previous sections, one sees that the concepts of indistinguishability and limits (in this section) and conditional expectation (in the next section) fit well into the framework of $\mathcal{RV}$ and $\mathcal{PR}$. All results for $\mathcal{RV}$ also carry to $\mathcal{PR}$ via the (countable) product property.

For this section, fix some admissible σ-algebra on Ω again.

Lemma 5.23. *The equivalence classes in $\mathcal{RV}\ X$ with respect to indistinguishability of maps as of remark 5.3 are exactly the atoms of $\mathcal{RV}\ X$. In particular, $\mathcal{RV}\ X$ is atomic.*

Proof. For $o \in \mathcal{RV}\ X$, let $[o]$ be the equivalence class of o with respect to indistinguishability. $[o]$ is measurable:

$$\begin{aligned}[o] &= \bigcap_{\omega\in\Omega} B_{\omega,K_{o(\omega)}} \\ &= \bigcap_{\omega\in\Omega} B_{K_\omega,K_{o(\omega)}}\end{aligned}$$

where the RHS is in fact a countable intersection as by admissibility, there are only countably many choices for K_ω, and K_ω determines $K_{o(\omega)}$.

To see that $[o]$ is an atom , I show that the set

$$\{B \subseteq \mathcal{RV}\ X \mid \forall o \in B : [o] \subseteq B\}$$

is a σ-algebra containing all the $B_{\omega,A}$.

- Let $\omega \in \Omega$ and $A \subseteq X$ measurable. Let o, o' be indistinguishable and $o \in B_{\omega,A}$, i.e. $o(\omega) \in A$. By the definition of indistinguishability, then also $o'(\omega) \in A$, i.e. $o' \in B_{\omega,A}$. Hence, $B_{\omega,A}$ has the property.
- The σ-algebra properties follow just like in the proof of lemma 5.7.1 in appendix C. □

Corollary 5.24. *Two measurable maps $o, o' : \Omega \to X$ are indistinguishable as maps iff they are indistinguishable as elements of $\mathcal{RV}\ X$.*

Proof. Two elements of an atomic space are indistinguishable iff they lie in the same atom. Now apply lemma 5.23. □

`curry` and `uncurry` provide means of going back and forth between maps from certain measurable spaces to X and maps to $\mathcal{RV}\ X$. This has interesting consequences for point-wise operations:

Lemma 5.25. *Let $X \in \mathcal{M}$ be atomic.*

The measurable maps $X \to \mathcal{RV}\ \mathbb{R}$ are closed under point-wise limits in the following sense:

Let $f_i : X \to \mathcal{RV}\ \mathbb{R}$ for $i \in \mathbb{N}$ be measurable maps.

1. *The set*

$$L_f := \left\{x \in X \mid \forall \omega \in \Omega : (f_i(x)(\omega))_{i\in\mathbb{N}} \text{ converges}\right\}$$

   is measurable in X.

2. *The map*

$$\begin{aligned}\lim_{i\to\infty} f_i : \quad & L_f \to \mathcal{RV}\ \mathbb{R} \\ \left(\lim_{i\to\infty} f_i\right)(x) := & \left(\omega \mapsto \lim_{i\to\infty} f_i(x)(\omega)\right)\end{aligned}$$

   is well-defined and measurable.

Proof. By lemma 5.14, the functions $\texttt{uncurry}\ f_i : X \times \Omega \to \mathbb{R}$ are measurable. Let

$$J_f = \left\{(x,\omega) \in X \times \Omega \mid ((\texttt{uncurry}\ f_i)(x,\omega))_{i\in\mathbb{N}} \text{ converges}\right\}.$$

By a standard theorem from measure theory,[36] J_f is measurable.

$x \in X$ is in L_f iff for all $\omega \in \Omega$, $(x,\omega) \in J_f$, so

$$L_f = (J_f)_\Omega.$$

By corollary 5.10.1, this is measurable, i.e. 1 holds.

To show 2, wlog. assume that $L_f = X$. Otherwise, replace X by its measurable subset L_f. Again by a standard theorem[12, thm. 21.3], the map $\lim_i (\texttt{uncurry}\ f_i) = ((x,\omega) \mapsto \lim_i f_i(x)(\omega))$ is measurable. Then by the other direction of lemma 5.14, $\lim_i f_i = \texttt{curry}\ (\lim_i (\texttt{uncurry}\ f_i))$ is well-defined and measurable. □

Remark 5.26. The statement of lemma 5.25 holds for sup, inf, lim sup etc. with the same proof.

5.2.5 Expectation

Fix now a common probability measure $\mathbb{P}$ on Ω for the σ-algebras mentioned below. This will amount to having $\mathbb{P}$ always be a probability measure on $\mathcal{B}$.

Definition 5.27. Let $\mathcal{A} \subseteq \mathcal{B}$ be two σ-algebras on Ω such that $(\Omega,\mathcal{A})$ and $(\Omega,\mathcal{B})$ are both admissible sample spaces. Define for $A, B \in \mathcal{B}$:

$$\mathbb{P}[A \mid B] := \begin{cases} 0 & \text{if } \mathbb{P}[B] = 0 \\ \frac{\mathbb{P}[A\cap B]}{\mathbb{P}[B]} & \text{otherwise} \end{cases}$$

For $o \in \mathcal{RV}^{\mathcal{B}}\ \mathbb{R}$ and $\omega \in \Omega$ define

$$\mathbb{E}[o \mid \mathcal{A}](\omega) := \sum_{L\in\mathcal{B}\text{ atom}} o(L) \cdot \mathbb{P}\left[L \mid K^{\mathcal{A}}_\omega\right]$$

if this countably infinite sum is well-defined.

Here, $o(L)$ is the unique value in the image of the set L under the function o. This is well-defined as o is measurable in $\mathcal{B}$, L is always an atom of $\mathcal{B}$ and points are measurable in $\mathbb{R}$.

Let $L^0(\mathcal{B},\mathcal{A})$ be the set of $o \in \mathcal{RV}^{\mathcal{B}}\ \mathbb{R}$ such that the above expression is well-defined for all ω.

In the above definition, the value of $\mathbb{P}[A \mid B]$ under $\mathbb{P}[B] = 0$ is arbitrary, of course. Lemma 5.32 will show that this is fine. The following lemma will show that the above function as well as its domain are measurable.

First, notice that the above definition includes the (unconditional) expectation as a special case:

[36] Let $g_i := \texttt{uncurry}\ f_i$. $y \in J_f$ iff $(g_i(y))_i$ converges, i.e. iff it is a Cauchy sequence. Hence, $J_f = \bigcap_{\varepsilon\in\mathbb{Q}^+} \bigcup_{N\in\mathbb{N}} \bigcap_{n,m\geq N} \{y \mid |g_n(y) - g_m(y)| < \varepsilon\}$.

Definition 5.28. In definition 5.27, let $\mathcal{A} = \{\emptyset, \Omega\}$ be the trivial σ-algebra on Ω. Then $\mathbb{E}[o \mid \mathcal{A}]$ – if it exists – is constant as Ω is an atom of $\mathcal{A}$. Let

$$\mathbb{E}[o]$$

be the unique value of $\mathbb{E}[o \mid \mathcal{A}]$.

Let $L^0(\mathcal{B})$ be the set of $o \in \mathcal{RV}^{\mathcal{B}}\ \mathbb{R}$ such that $\mathbb{E}[o]$ exists.

Theorem 5.29. *Let $\mathcal{A}$ and $\mathcal{B}$ be as in definition 5.27.*

1. *If $L \in \mathcal{B}$ is an atom and $\omega \in \Omega$, define*

$$p_L(\omega) := \mathbb{P}\left[L \mid K_\omega^{\mathcal{A}}\right].$$

   $p_L \in \mathcal{RV}^{\mathcal{A}}\ \mathbb{R}$.

2. *$L^0(\mathcal{B}, \mathcal{A}) \subseteq \mathcal{RV}^{\mathcal{B}}\ \mathbb{R}$ is measurable.*

3. *The map*

$$\mathbb{E}[\cdot \mid \mathcal{A}] : L^0(\mathcal{B}, \mathcal{A}) \to \mathcal{RV}^{\mathcal{A}}\ \mathbb{R}$$

   is well-defined and measurable.

Proof. 1: Let $K_L^{\mathcal{A}}$ be the unique atom of $\mathcal{A}$ containing L. This must exist as the atoms of $\mathcal{A}$ are pairwise-disjoint $\mathcal{B}$-measurable sets covering Ω and L is an atom of $\mathcal{B}$. For the same reason, $L \cap K_\omega^{\mathcal{A}}$ is either L or $\emptyset$ for any $\omega \in \Omega$ and it is L iff $K_L^{\mathcal{A}} = K_\omega^{\mathcal{A}}$, i.e. iff $\omega \in K_L^{\mathcal{A}}$. Hence, we have

$$p_L = \begin{cases} \mathbb{P}[L \mid K_L^{\mathcal{A}}] & \text{on } K_L^{\mathcal{A}} \\ 0 & \text{anywhere else,} \end{cases}$$

which is clearly $\mathcal{A}$-measurable.

2, 3: I apply lemma 5.25 to the partial sums in the definition of $\mathbb{E}[\cdot \mid \mathcal{A}]$.

Fix some enumeration[37] $\{L_i \mid i \in \mathbb{N}\}$ of the atoms of $\mathcal{B}$. Define for $j \in \mathbb{N}$ and $o \in \mathcal{RV}^{\mathcal{B}}\ \mathbb{R}$:

$$f_j(o) := \sum_{i=0}^{j} o(L_i) \cdot p_{L_i}$$

For any j, $f_j : \mathcal{RV}^{\mathcal{B}}\ \mathbb{R} \to \mathcal{RV}^{\mathcal{A}}\ \mathbb{R}$ is a well-defined and measurable function:

- For any i and o, $o(L_i) \cdot p_{L_i} \in \mathcal{RV}^{\mathcal{A}}\ \mathbb{R}$ as p_{L_i} is $\mathcal{A}$-measurable by part 1 and $o(L_i)$ is a constant.
- For any i, the map $(o \mapsto o(L_i) \cdot p_{L_i})$ is measurable as $(o \mapsto o(L_i))$ is measurable by choice of the σ-algebra on $\mathcal{RV}^{\mathcal{B}}\ \mathbb{R}$ and p_{L_i} is a constant with respect to o.
- The finite sum corresponds to a lift applied on top of the functions $(o \mapsto o(L_i) \cdot p_{L_i})$, so f_j is well-defined and measurable as well.

[37] assuming wlog. that $\mathcal{B}$ is infinite

Seeing that

$$\mathbb{E}\left[o \mid \mathcal{A}\right](o)(\omega) = \lim_{j\to\infty} f_j(o)(\omega),$$

the claim now follows from lemma 5.25: In the definition from there, $L^0\left(\mathcal{B},\mathcal{A}\right) = L_f$ and $\mathbb{E}[\cdot \mid \mathcal{A}] = \lim_{j\to\infty} f_j$. Lemma 5.25 can in fact be applied here as $\mathcal{RV}^{\mathcal{B}}\ \mathbb{R}$ is atomic by lemma 5.23. □

Definition 5.30. Let $\mathcal{A}$ and $\mathcal{B}$ be as in definition 5.27 and $p \in \mathbb{N}$, $p \geq 1$. Define the following sets:

$$L^p\left(\mathcal{A},\mathcal{B}\right) := \left\{o \in \mathcal{RV}^{\mathcal{B}}\ \mathbb{R} \mid |o|^p \in L^0\left(\mathcal{B},\mathcal{A}\right)\right\}$$
$$L^p\left(\mathcal{B}\right) := \left\{o \in \mathcal{RV}^{\mathcal{B}}\ \mathbb{R} \mid |o|^p \in L^0\left(\mathcal{B}\right)\right\}$$

Here, $|o|^p$ denotes $\texttt{fmap}\ (x \mapsto |x|^p)\ o$, of course.

Corollary 5.31. *The sets $L^p\left(\mathcal{A},\mathcal{B}\right) \subseteq \mathcal{RV}^{\mathcal{B}}\ \mathbb{R}$ are all measurable.*

Proof. $L^p\left(\mathcal{A},\mathcal{B}\right)$ is the preimage under the measurable function (lift) $(o \mapsto |o|^p)$ of the, by theorem 5.29, measurable set $L^0\left(\mathcal{B},\mathcal{A}\right)$. □

So definition 5.27 of the conditional expectation has all the properties one would naturally want. It remains to show that this definition is actually correct.

Lemma 5.32. *Let $\mathcal{A}$, $\mathcal{B}$, and o be as in definition 5.27.*

$\mathbb{E}[o \mid \mathcal{A}]$ is (almost surely) the conditional expectation of the real-valued random variable o given the σ-algebra $\mathcal{A}$.

Proof. Let $\mathbb{E}'\left[p\right]$ be the "actual" expectation, i.e. the integral of a random variable p. It is easy to see that for $o \in \mathcal{RV}^{\mathcal{A}}\ \mathbb{R}$

$$\mathbb{E}'\left[o\right] = \sum_{K\in\mathcal{A}\ \text{atom}} o(K)\cdot\mathbb{P}\left[K\right] \tag{5.1}$$

if any of the two sides exists.

One needs to show that for any $A \in \mathcal{A}$ the following holds:[38]

$$\mathbb{E}'\left[\mathbb{E}\left[o \mid \mathcal{A}\right]\cdot\mathbb{1}_A\right] = \mathbb{E}'\left[o\cdot\mathbb{1}_A\right] \tag{5.2}$$

where $\mathbb{1}_A$ is the characteristic function of A.

Note that $\mathbb{E}[o \mid \mathcal{A}]\cdot\mathbb{1}_A$ is $\mathcal{A}$-measurable while $o\cdot\mathbb{1}_A$ is only $\mathcal{B}$-measurable in general.

Inserting the definition of $\mathbb{E}[o \mid \mathcal{A}]$ and equation (5.1), one receives that the LHS is equal to

$$\sum_{\substack{K\in\mathcal{A}\\ \text{atom}}}\ \sum_{\substack{L\in\mathcal{B}\\ \text{atom}}} \mathbb{1}_A(K)\cdot o(L)\cdot\mathbb{P}\left[L \mid K\right]\cdot\mathbb{P}\left[K\right].$$

In the above sum, $\mathbb{1}_A(K) = 0$ unless $K \subseteq A$ (and otherwise $\mathbb{1}_A(K) = 1$) and $\mathbb{P}[L \mid K] = 0$ unless $L \subseteq K$. Also note that for $L \subseteq K$, $\mathbb{P}[L \mid K]\cdot\mathbb{P}[K] = \mathbb{P}[L]$,

[38] cf. e.g. [2, p. 404]

even for the special case $\mathbb{P}[K] = 0$ as then also $\mathbb{P}[L] = 0$. So the above sum is equal to

$$\begin{aligned}
&\sum_{\substack{K \in \mathcal{A} \text{ atom} \\ K \subseteq A}} \sum_{\substack{L \in \mathcal{B} \text{ atom} \\ L \subseteq K}} o(L) \cdot \mathbb{P}[L] \\
&= \sum_{\substack{L \in \mathcal{B} \text{ atom} \\ L \subseteq A}} o(L) \cdot \mathbb{P}[L] \\
&= \sum_{L \in \mathcal{B} \text{ atom}} \mathbb{1}_A(L) \cdot o(L) \cdot \mathbb{P}[L] \\
&= \mathbb{E}'[o \cdot \mathbb{1}_A]
\end{aligned}$$

as required. Here, the second line follows as the atoms K of $\mathcal{A}$ define a partition of Ω and therefore, being $\mathcal{B}$-measurable sets, also of the atoms L of $\mathcal{B}$. The last line is again (5.1).

From the above considerations it is further clear that the LHS exists iff the RHS exists. This concludes the proof. □

Corollary 5.33. *The unconditional expectation $\mathbb{E}[\cdot]$ from definition 5.28 is well-formed in the following sense: Let $(\Omega, \mathcal{B})$ be an admissible sample space.*

1. *$L^0(\mathcal{B}) \subseteq \mathcal{RV}^{\mathcal{B}}\ \mathbb{R}$ is measurable.*
2. *The map*
$$\mathbb{E}[\cdot] : L^0(\mathcal{B}) \to \mathbb{R}$$
is well-defined and measurable.
3. *The sets $L^p(\mathcal{B}) \subseteq \mathcal{RV}^{\mathcal{B}}\ \mathbb{R}$ are all measurable.*
4. *If $o \in \mathcal{RV}^{\mathcal{B}}\ \mathbb{R}$, then $\mathbb{E}[o]$ is the expectation of the real-valued random variable o.*

Proof. These all follow as special cases of theorem 5.29, corollary 5.31 and lemma 5.32 when choosing the trivial σ-algebra for $\mathcal{A}$.

For 2, we receive that the map

$$\mathbb{E}[\cdot \mid \{\emptyset, \Omega\}] : \mathcal{RV}^{\mathcal{B}}\ \mathbb{R} \to \mathcal{RV}^{\{\emptyset,\Omega\}}\ \mathbb{R}$$

is measurable. And $\mathcal{RV}^{\{\emptyset,\Omega\}}\ \mathbb{R}$ is isomorphic to $\mathbb{R}$ via evaluation at the atom Ω and its inverse `return`.

For 4, apply (5.2) to $A = \Omega$. Inserting (5.1) at the LHS, it is clear that the expectation of o corresponds to evaluation at the single atom Ω of $\{\emptyset, \Omega\}$. □

As a last step, one can consider $\mathcal{PR}$ again: Let $\mathbb{T}$ and $(\mathcal{F}_t)_{t \in \mathbb{T}}$ be as above, let $\mathbb{P}$ be a common probability measure for all the $\mathcal{F}_t$ and let Δt be a positive number. Let

$$\mathcal{PR}_{\Delta t} := \bigtimes_{\substack{t \in \mathbb{T} \\ t + \Delta t \in \mathbb{T}}} L^0(\mathcal{F}_{t+\Delta t}, \mathcal{F}_t)$$

and define

$$\begin{aligned}
\texttt{shift}_{\Delta t} :&\quad \mathcal{PR}_{\Delta t}\ \mathbb{R} \to \mathcal{PR}\ \mathbb{R} \\
(\texttt{shift}_{\Delta t}\ o)_t &:= \begin{cases} \mathbb{E}[o_{(t+\Delta t)} \mid \mathcal{F}_t] & \text{if } t + \Delta t \in \mathbb{T} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Lemma 5.34. *The function* $\texttt{shift}_{\Delta t}$ *as defined above is well-defined and measurable.*

Proof. $\texttt{shift}_{\Delta t}$ is the combination of the functions

$$g_t : \quad \mathcal{PR}_{\Delta t}\ \mathbb{R} \xrightarrow{\text{proj}} L^0\left(\mathcal{F}_{t+\Delta t}, \mathcal{F}_t\right) \xrightarrow{\mathbb{E}[\,\cdot\,|\mathcal{F}_t]} \mathcal{RV}^{\mathcal{F}_t}\mathbb{R}$$

for $t+\Delta t \in \mathbb{T}$ and $g_t := \texttt{const}\ (\texttt{return}\ 0)$ for $t+\Delta t \notin \mathbb{T}$. By the above theorems, these are all well-defined and measurable and then the statement follows from the universal property of the product. □

Remark 5.35. If $((\mathcal{F}_t)_{t\in\mathbb{T}})$ is such that the conditional expectation always exists, i.e. such that $L^0\left(\mathcal{F}_{t+1}, \mathcal{F}_t\right) = \mathcal{RV}^{\mathcal{F}_{t+1}}\ \mathbb{R}$ for all t with $t + \Delta t \in \mathbb{T}$, then one can interpret

$$\texttt{shift}_{\Delta t} : \quad \mathcal{PR}\ \mathbb{R} \xrightarrow{\text{proj}} \mathcal{PR}_{\Delta t}\ \mathbb{R} \xrightarrow{\texttt{shift}_{\Delta t}} \mathcal{PR}\ \mathbb{R}.$$

This will be used in the following section 5.3

5.3 Modeling contracts by their present value

In the following, I will build a model for the `Con` type, therewith completing the probabilistic model of LPT.

Notation 5.36. I will again leave out "$\cdot^{\mathscr{A}}$" in most cases for the ease of reading.

Consider a special case of the model for the LPT_{Prim} and LPT_{Obs} parts from above where the additional assumption is made that

1. $\mathbb{T} = \{1, \ldots, T\}$ for some $T \in \mathbb{N}$ and
2. $\mathcal{F}_t$ is finite for each t.

Fix a common probability measure $\mathbb{P}$ for the $\mathcal{F}_t$.

Remark 5.37. The canonical form (cf. remarks 5.3 and 5.20) of such a filtration is a finite tree.

As mentioned before, I want to use $\texttt{Con}^{\mathscr{A}} = (\texttt{Obs}\ \mathbb{R})^{\mathscr{A}} = \mathcal{PR}\ \mathbb{R}$.

Choose some set with the discrete σ-algebra as a model for the `Currency` type. Fix a *valuation currency* $K \in \texttt{Currency}$ and choose an observable $R \in \texttt{Obs}\ \mathbb{R}^+ = \mathcal{PR}\ \mathbb{R}^+$ and for any $k \in \texttt{Currency}$ an observable $W^{K/k} \in \mathcal{PR}\ \mathbb{R}^+$ such that the following hold:

$$R > 0 \tag{5.3}$$
$$W^{K/k} > 0 \tag{5.4}$$
$$W^{K/K} = 1 \tag{5.5}$$

R will describe the one-period-K-interest rate, i.e. $\frac{1}{R}$ will be the price of a zero-coupon bond over one unit of currency K with maturity one time step[39]

[39] except for R_T (= R at time T), which is ignored. The interest rate, as of section 4.2, must be ∞ at time T.

and $W^{K/k}$ is the K/k-exchange rate, i.e. the number of units of currency K corresponding to one unit of currency k (cf. sections 4.2 and 4.3).

In other words, we will have in $\mathscr{A}$:

$$\begin{aligned} \texttt{after}\ 1\ (\texttt{one}\ K) &\approx \frac{1}{R} \cdot \texttt{one}\ K \\ \texttt{one}\ k &\approx W^{K/k} \cdot \texttt{one}\ K \end{aligned}$$

`after` is from figure 7 in section 3.

Example 5.38. $R = 1$ yields a model without interest, i.e. $\texttt{after}\ \Delta t\ (\texttt{one}\ K) \approx \texttt{one}\ K$ for any $\Delta t :: \texttt{TimeDiff}$.

The meaning of $x \in \texttt{Con} = \texttt{Obs}\ \mathbb{R} = \mathcal{PR}\ \mathbb{R}$ will be that for $t \in \mathbb{T}$, x_t is the present value in currency K and at time t of the contract x, expressed as a random variable in the information available at time t (which is described by $\mathcal{F}_t$). This is equivalent to saying that at time t, receiving a payment of x_t units of K is equally preferable to acquiring x. The interest rate R will be used to discount future payments.

Note how for $x, y \in \texttt{Con}$, one can construct $(x \leq y) = (\texttt{lift}_2\ (\leq)\ x\ y) \in \mathcal{PR}\ \texttt{Bool} = \texttt{Obs}\ \texttt{Bool} = \texttt{OB}$. Define for $\mathscr{A}$:

$$x \preceq_b y \ :\Leftrightarrow\ b \Rightarrow x \leq y$$

Note how $x \approx y$ iff $x = y$ in this model. One can already see that the logical axioms from section 3.1.1 hold in $\mathscr{A}$ by fixing certain $\omega \in \Omega$ and $t \in \mathbb{T}$: Then one only needs to consider the operations "$\vee$", "$\rightarrow$" etc. on `Bool`.

The last remaining definitions are the primitive operations on contracts:

5.3.1 The time-local primitives

Model $\texttt{one}\ k$ by the fixed exchange rates from above:

$$\texttt{one}\ k := W^{K/k}$$

Note how now, $x \in \texttt{Con}$ is its own price in currency K as of definition 4.1.

Note further how the axiom $\texttt{zero} \prec \texttt{one}$ follows because we required that $W^{K/k} > 0$ for any currency. (5.5) is just a normalizing condition here that ensures that $\texttt{one}\ K$ has always present value 1.

Theorem 4.3 now dictates that the models of `zero`, `give`, `and`, `scale` and `or` must be point-wise application of 0, $(-)$, $(+)$, multiplication with a constant and maximum, respectively. So choose the interpretations like that. From the same theorem, one receives that "$\rightsquigarrow$" must be "$\gg\!=$" and so $\texttt{read}'$ must be `join`, so choose this. We have already seen in the discussion of $\mathcal{PR}$ that these maps are well-defined and measurable.

It is easy to see that the laws from section 3.1 which involve only these time-local primitives hold in $\mathscr{A}$, except for maybe the rule for "$\rightsquigarrow$" is not completely obvious:

Lemma 5.39. *If $a, b \in \mathcal{M}$, $o_1 \in \mathcal{PR}\ a$, $o_2 \in \mathcal{PR}\ b$, $f_1 : a \to \mathcal{PR}\ \mathbb{R}$ and $f_2 : b \to \mathcal{PR}\ \mathbb{R}$ are measurable functions and $d \in \mathcal{PR}\ \texttt{Bool}$, then the following axiom for "$\rightsquigarrow$" holds in $\mathscr{A}$:*

$$\left(\forall x_1 :: a_1,\, x_2 :: a_2 :\ \underset{d \wedge o_1 = x_1 \wedge o_2 = x_2}{f_1\ x_1 \preceq f_2\ x_2}\right) \Rightarrow o_1 \rightsquigarrow f_1\ \preceq_d\ o_2 \rightsquigarrow f_2 \quad (*3.38)$$

Proof. The premise is in $\mathscr{A}$ equivalent to

$$(\lambda\, x_1\ x_2.\ ((d \wedge o_1 = x_1 \wedge o_2 = x_2) \to f_1\ x_1 \leq f_2\ x_2)) = \mathtt{const}_2\ \top.$$

Now the rest of the proof can be done inside LPT: By lift reduction

$$((o_1, o_2) \gg\!= \lambda\ (x_1, x_2)\,.\,(d \to f_1\ x_1 \leq f_2\ x_2)) = \top.$$

d inside the lambda term does not depend on x_1 or x_2, so this is easily seen to imply

$$d \Rightarrow ((o_1, o_2) \gg\!= \lambda\ (x_1\ x_2)\,.\,(f_1\ x_1 \leq f_2\ x_2))$$

and the RHS is equal to

$$(o_1 \gg\!= f_1) \leq (o_2 \gg\!= f_2),$$

so we receive in $\mathscr{A}$ the required conclusion $o_1 \rightsquigarrow f_1 \preceq_d o_2 \rightsquigarrow f_2$. □

So left are $\mathtt{when}'$ and `anytime`.

5.3.2 $\mathtt{when}'$ and `anytime`

The easiest way do define these is to give an interpretation of the `next` operation from section 3.4. Recap that we defined

$$\mathtt{next}\ x = \mathfrak{n} \rightsquigarrow \lambda t.\ \mathfrak{W}'\ (\mathfrak{n} > t)\ x. \tag{5.6}$$

In $\mathscr{A}$, time is discrete, so `next` x can be thought to mean "acquire x after one time step". $(\mathtt{next}\ x)_t$ should hence be the present value, expressed as a $\mathcal{F}_t$-random variable, of acquiring x at time $t+1$ (unless $t = T$, then it is 0). This can be computed as follows:

1. Take the conditional expectation of x_{t+1} under $\mathcal{F}_t$, i.e. the expected value of acquiring x at time $t+1$ given the information at time t. The conditional expectation always exists as $\mathcal{F}_{t+1}$ is finite.

   This is achieved for all t by the $\mathtt{shift}_{\Delta t}$ function from the end of the previous section 5.2.5 for $\Delta t = 1$.

2. The result is thought to be a payment at time $t+1$, so use the provided interest rate R_t to discount it one time step back.

   This is achieved for all t by multiplying with $\frac{1}{R}$. Recap that we required $R > 0$, so this is well-defined.

Translated into formulas, one receives

$$\mathtt{next}\ x := \frac{1}{R} \cdot \mathtt{shift}_1\ x \tag{5.7}$$

which expands to

$$\begin{aligned}(\mathtt{next}\ x)_T &:= 0\\ (\mathtt{next}\ x)_t &:= \frac{1}{R_t} \cdot \mathbb{E}\left[x_{t+1} \mid \mathcal{F}_t\right] \qquad \text{for } t < T.\end{aligned}$$

Figure 10 Definition of `when′` and `anytime` in $\mathcal{A}$ resulting from (3.39), (3.40) and (5.7)

$$
\begin{aligned}
(\texttt{when}'\ b\ x)_T &:= \texttt{if}'\ (\mathfrak{e}\ b)_T\ x_T\ 0 \\
(\texttt{when}'\ b\ x)_t &:= \texttt{if}'\ (\mathfrak{e}\ b)_t\ x_t\ \left(\frac{1}{R_t}\cdot\mathbb{E}\left[\left(\texttt{when}'^{\mathcal{A}}\ b\ x\right)_{t+1} \mid \mathcal{F}_t\right]\right) \\
&\quad \text{for } t < T \\
(\texttt{anytime}\ x)_T &:= \max(x_T,\, 0) \\
(\texttt{anytime}\ x)_t &:= \max\left(x_t,\, \frac{1}{R_t}\cdot\mathbb{E}\left[(\texttt{anytime}\ x)_{t+1} \mid \mathcal{F}_t\right]\right) \\
&\quad \text{for } t < T
\end{aligned}
$$

Here, $\texttt{if}'\ c\ x\ y$ should mean the lift $(\omega \mapsto \texttt{if}'\ (c(\omega), x(\omega), y(\omega)))$ of the function $\texttt{if}' : \texttt{Bool} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ to $\mathcal{RV}^{\mathcal{F}_t}$ and likewise max should mean the lift of the maximum function.

Here, the RHSs are expressions in $\mathcal{RV}^{\mathcal{F}_t}\ \mathbb{R}$. It was seen previously that $\texttt{shift}_1$, and hence `next`, is well-defined and measurable.

Note that it is not yet clear that this definition of `next` is sensible. I will show that `when′` as defined right below fulfills the axioms and yields this definition of `next`.

However, if one just assumes that the definition makes sense, one immediately receives the unique possible definitions of `when′` and `anytime` from section 3.4: We need in abstract terms

$$\mathfrak{W}'\ b\ x \approx \texttt{cond}\ (\mathfrak{e}\ b)\ x\ (\texttt{next}\ (\mathfrak{W}'\ b\ x)) \tag{3.39}$$

$$\mathfrak{A}\ x \approx x \vee \texttt{next}\ (\mathfrak{A}\ x) \tag{3.40}$$

and these then directly yield the inductive definitions of `when′` and `anytime` as depicted in figure 10.

The interpretation of `anytime` is also called the *Snell envelope* of the stochastic process x. Peyton Jones and Eber [3] mention the Snell envelope as the suitable model for `anytime`. For a discussion of the structural properties of the Snell envelope cf. [2, p. 280].

Lemma 5.40. `when′` *and* `anytime` *as defined above are well-defined and measurable maps*

$$\begin{aligned}
\texttt{when}' &: \mathcal{PR}\ \texttt{Bool} \times \mathcal{PR}\ \mathbb{R} \to \mathcal{PR}\ \mathbb{R} \\
\texttt{anytime} &: \mathcal{PR}\ \mathbb{R} \to \mathcal{PR}\ \mathbb{R}.
\end{aligned}$$

Proof. These maps can be constructed by finitely many applications of `next` and lifts. For example, for $T = 3$ we have

$$\texttt{anytime}\ x\ =\ x \vee \texttt{next}\ (x \vee \texttt{next}\ (x \vee 0)).$$

Here, "$\vee$" is the interpretation of `or` which is already known to be measurable. □

As a first indicator of the correctness of $\texttt{when}'$, note how the abstract definition (5.6) corresponds to the definition in $\mathscr{A}$ (5.7) now and how (3.39) and (3.40) hold.

Still assuming that the given definition of $\texttt{when}'$ conforms to the axioms, our definition of `anytime` does so as well: It fulfills the recursive equation (3.40), $\mathscr{A}$ has reverse inductive time and so by theorem 3.47 `anytime` is correct: The proof of this theorem did not use the existence of `anytime` and showed that any contract for which this equation holds has the required property.

So all that's left is to show that $\texttt{when}'$ conforms to the axioms:

As a first block, one needs to show:

$$\texttt{when}'\ b\ 0 \approx 0 \tag{*3.20}$$

$$\texttt{when}'\ b\ (x+y) \approx \texttt{when}'\ b\ x + \texttt{when}'\ b\ y \tag{*3.21}$$

$$\texttt{when}'\ b\ (\alpha \cdot x) \approx \alpha \cdot (\texttt{when}'\ b\ x) \tag{*3.22}$$

Considering the definition of $\texttt{when}'$, these follow directly (as usual, via downwards induction on $t \in \mathbb{T}$) by linearity of the involved operations $(\texttt{if}'\ (\mathfrak{e}\ b)_t)$, scaling by the random variables R_t and conditional expectation.

Now towards the behavior of $\texttt{when}'$ over time. To show:

$$\mathfrak{W}'\ b\ x \approx_{\mathfrak{e}\ b} x \tag{*3.23}$$

$$x \underset{\mathfrak{e}\ d\wedge b}{\preceq} \mathfrak{W}'\ c\ y \text{ and } \mathfrak{W}'\ b\ x \underset{\mathfrak{e}\ d\wedge c}{\preceq} y \Rightarrow \mathfrak{W}'\ b\ x \underset{\mathfrak{e}\ d\wedge\neg\mathfrak{e}\ b\wedge\neg\mathfrak{e}\ c}{\preceq} \mathfrak{W}'\ c\ y \tag{*3.24}$$

Lemma 5.41. *Axioms* (*3.23) *and* (*3.24) *hold in* $\mathscr{A}$.

Proof. In the following proof, I will leave out the sample parameter $\omega \in \Omega$, the time $t \in \mathbb{T}$ and – as before – the interpretation marker $\cdot^{\mathscr{A}}$ where possible.

(*3.23) means in $\mathscr{A}$ that

$$\mathfrak{e}\ b \Rightarrow (\texttt{when}'\ b\ x = x).$$

This is clear by definition of $\texttt{when}'$ as it always contains a "$\texttt{if}'\ (\mathfrak{e}\ b)_t\ x_t$" in front for any t.

For (*3.24), assume wlog. that $d = \mathfrak{e}\ d$ and assume that the premise holds. I.e., using the definition of "$\preceq_\cdot$" in $\mathscr{A}$ we assume:

a) $d \wedge b \Rightarrow x \leq \mathfrak{W}'\ c\ y$

b) $d \wedge c \Rightarrow \mathfrak{W}'\ b\ x \leq y$

One needs to show that for any $t \in \mathbb{T}$, whenever d_t is true and $(\mathfrak{e}\ b)_t$ and $(\mathfrak{e}\ c)_t$ are false, then

$$(\mathfrak{W}'\ b\ x)_t \leq (\mathfrak{W}'\ c\ y)_t.$$

I show this via backwards induction on t.

Consider in the following states where $d_t \wedge \neg(\mathfrak{e}\ b)_t \wedge \neg(\mathfrak{e}\ c)_t$ is true.

If $t = T$, then by definition $(\mathfrak{W}'\ b\ x)_t = 0 = (\mathfrak{W}'\ c\ y)_t$. So assume $t < T$ and assume that the statement holds for $t+1$.

The following are true as well:

i. d_{t+1} because $d = \mathfrak{e}\ d$.

ii. $(\mathfrak{e}\ b)_{t+1} \leftrightarrow b_{t+1}$ and $(\mathfrak{e}\ c)_{t+1} \leftrightarrow c_{t+1}$ because $\mathfrak{e}\ b$ and $\mathfrak{e}\ c$ were false at time t.

iii. "$\mathfrak{W}'$" reduces to its second case:

$$(\mathfrak{W}'\ b\ x)_t = \frac{1}{R_t}\mathbb{E}\left[(\mathfrak{W}'\ b\ x)_{t+1} \mid \mathcal{F}_t\right]$$
$$(\mathfrak{W}'\ c\ y)_t = \frac{1}{R_t}\mathbb{E}\left[(\mathfrak{W}'\ c\ y)_{t+1} \mid \mathcal{F}_t\right]$$

By iii., it suffices to show that $(\mathfrak{W}'\ b\ x)_{t+1} \leq (\mathfrak{W}'\ c\ y)_{t+1}$. – The conditional expectation and scaling are monotonic, of course.

By i. and ii., the following is a case distinction (with overlaps) for time $t+1$ over all states considered above:

- If $d_{t+1} \wedge \neg b_{t+1} \wedge \neg c_{t+1}$ holds, then by the induction hypothesis, $(\mathfrak{W}'\ b\ x)_{t+1} \leq (\mathfrak{W}'\ c\ y)_{t+1}$.
- If $d_{t+1} \wedge b_{t+1}$ holds, then $(\mathfrak{W}'\ b\ x)_{t+1} = x_{t+1}$ by definition. And $x_{t+1} \leq (\mathfrak{W}'\ c\ y)_{t+1}$ by a).
- If $d_{t+1} \wedge c_{t+1}$ holds, the statement follows analogously from b). □

Altogether one receives:

Theorem 5.42. *$\mathscr{A}$ as defined above is a model of* LPT.

Corollary 5.43. LPT *is consistent if set theory is.*

Remark 5.44. I only required existence of the interest factor for a single period and currency K. However, as any contract has a price, all interest factors exist. In fact, the interest factor $R_{b,k}$ is *equal* to

$$\frac{1}{\mathfrak{W}'\ b\ (\texttt{one}\ k)}$$

in $\mathscr{A}$.

Remark 5.45. The model can be used to compute present values as numbers by setting $\mathcal{F}_1 = \{\emptyset, \Omega\}$. Then $\mathcal{F}_1$-random variables are just constants.

6 Conclusion and outlook

I have established a probability-free, purely formal framework to model the arbitrage behavior of a large class of financial contracts. I have shown that the framework is sufficient to prove key statements from arbitrage theory and that a simple stochastic model can implement it. My approach shows how assumptions commonly made – such as a fixed interest rate – are not actually needed and makes other assumptions – such as the defining property of a dividend-free share – explicit.

The framework replaces complex portfolio arguments by a series of small, intuitively clear steps, related by the proven mathematical framework of many-sorted first-order logic. As LPT proofs are usually general, the interconnection and the deeper reasons for why certain arbitrage arguments work can be seen much clearer than when showing arbitrage relations for specific assets such as swaps or stock options. A possible application is therefore in teaching: A student trained with (a simplified version of) LPT will easily recognize the patterns present in real-world contracts. Teaching can clearly separate arbitrage statements and stochastic models and the transition to stochastics is made easy by giving implementations of the combinators.

Another application is where the framework [3] originally came from, namely in software (cf. below).

6.1 Future work

Models Further research should focus on whether more general models, such as infinite time and/or infinite states, can implement LPT and whether models with continuous states and time – such as the Black-Scholes model – can be applied: Aumann's work shows that evaluation is not possible in these cases, but it is not clear whether the monadic `join` can still be defined on a suitable σ-algebra without using evaluation.

BID-ASK spread Another question is how the assumption of perfect markets can be lifted to receive weaker arbitrage bounds that can include transaction costs or taxes. This ultimately amounts to handling the BID-ASK spread: As briefly discussed in section 4.3, in practice, there is not "the" price of an asset, but the price for buying (*ASK*) is slightly higher than that for selling (*BID*).[40] The difference is called the *spread*. It is clear that the (effective) spread, i.e. the difference between the price the buyer pays and the amount the seller receives, must be at least the total cost of the transaction.

The first question would be how to define BID- and ASK prices in the framework. I propose the following:

> o :: `Obs` $\mathbb{R}$ is called a *BID price* (*ASK price*) for x :: `Con` if o is maximal (minimal) with the property that $o \preceq x$ ($o \succeq x$).

This definition looks promising because it is a true generalization of the definition of a price from section 4.1: If x has a price, then that is both a BID- and

[40] It cannot be the other way round, otherwise buying, then directly selling would be an arbitrage strategy.

ASK price. Also, the "priceless" contract from example 4.14 has neither a BID- nor a ASK price, as is easily seen.

Note however that a BID-ASK spread introduces a considerable complication in that a contract y can be "priced higher" than a contract x in three different ways:

$\mathrm{ASK}(x) \leq \mathrm{BID}(y)$: y can always be exchanged for x without additional cost. Implies the other two.

$\mathrm{BID}(x) \leq \mathrm{BID}(y)$ and $\mathrm{ASK}(x) \leq \mathrm{ASK}(y)$: y is valued at least as high as x by the market, but we cannot in general exchange. Implies the third.

$\mathrm{BID}(x) \leq \mathrm{ASK}(y)$: One cannot make profit from buying y and selling x, i.e. exchanging x for y.

So even if there are BID- and ASK prices for the two contracts, it is not clear what a counterpart of lemma 3.12 for BID- and ASK prices should look like. To which extent the primitives are compatible to which of the above three relations is a subject of future research.

Value quantification for `OB` The primitives $\mathfrak{e}$ and $\mathfrak{a}$ allow quantification only over time and only in a limited way. It would be interesting to see how one can arrive at a stronger notion of quantification that makes patterns like "b has happened before, hence there must be a point in time where it was true" possible. A first idea is to introduce a combinator

$$\texttt{exists} :: (a \to \texttt{OB}) \to \texttt{OB}$$

where `exists` f is true whenever there is an $x :: a$ such that $f\ x$ is currently true. Together with reasonable axioms such as that `exists` commutes with "$\gg\!=$", one could show axiom (*2.4) from the others. But there are two problems:

1. From `exists`, one could make the solution to difficult mathematical problems a basis for contracts, such as "if a certain polynomial (read from some observable, of high degree) has a root, then receive a dollar, otherwise pay a dollar". This might well not be desired.

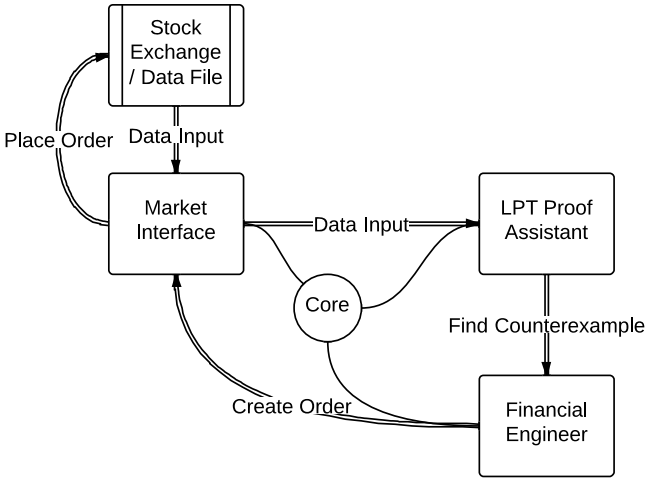
2. More critically, the canonical model of `exists` in the probabilistic model of observables from section 5.2 is for $f : X \to \texttt{Bool}$ the following:

$$\texttt{exists}\ f : \quad \mathcal{PR}\ \texttt{Bool}$$
$$(\texttt{exists}\ f)_t(\omega) := \begin{cases} \texttt{True} & \text{if there is some } x \in X \text{ with } f(x)(\omega) = \texttt{True} \\ \texttt{False} & \text{otherwise.} \end{cases}$$

   Even if $\mathbb{T}$ is a single point, it is easy to see that the preimage of $\{\texttt{True}\}$ under `exists` f is essentially a projection and hence not in general measurable.

Unobservables While LPT only talks about observables which are, by name, known to everyone, in reality not all events are observable. For example, a participant might choose to exercise a `anytime` option at the moment an urgent order from a foreign country arrives at her company. This event is not observable

Figure 11 LPT trader high-level overview



to the market, but it is the basis for a choice which *is* visible to the market. Another example would be insider information (which, however, violated the assumption of perfect markets).

Such unobservable events could be modeled by a new type constructor `UnObs` which is similar in structure to `Obs`, to which `Obs` embeds and which can occur as a condition in "$\preceq_.$, but which *cannot* occur as an argument to `when′`. Further assumptions could then describe the "degree of perfect information" the market has.

Algorithmic trading / market analysis LPT describes the behavior of perfect markets. In reality, however, markets are not perfect and arbitrage is possible for short periods of time. In fact, the assumption of arbitrage-freeness is based on the assumption that there *are* traders exploiting arbitrage opportunities as soon as they arise.

A question that arises now is the following: If arbitrage opportunities manifest as inconsistencies with LPT, then how can one use LPT instead of *describing* an arbitrage-free world to *identify* arbitrage opportunities and how would an *algorithmic trader*, i.e. a computer program trading at the stock exchange, use this capability? My hope is that a LPT-based trader could execute not only certain pre-defined strategies, but analyze the whole of the market to identify opportunities a conventional trader would not notice. A high-level overview is given in figure 11.

Even if the trader component is left out, LPT could be used to holistically analyze a derivatives market, which might be useful for research.

A Lambda notation and Haskell for many-sorted first-order logic

In this section, I describe in detail the formal framework in which all argumentation in this thesis happens.

As mentioned in section 1, this thesis is based on a paper about Haskell[5] and hence, notational and conceptual conventions were adopted from that programming language. The latter include a style generally centered around functions as in

$$(\gg\!=) :: \texttt{Obs}\ a \to (a \to \texttt{Obs}\ b) \to \texttt{Obs}\ b$$

from section 2. One would then use "$\gg\!=$" as in

$$o \gg\!= \lambda x.\ \texttt{return}\ (x + 1)$$

where "$\lambda x.\ \ldots$" should describe a function in one argument x.

The question is now what exactly this notation is supposed to mean: While arguing with functions can be done in an intuitive manner, my approach is axiomatic, and the notion of an „axiom" only makes sense inside a formal logic framework.

Let's call this style of writing functions "lambda notation". The first aim of this section is to give a solid meaning to lambda notation inside many-sorted first-order logic (*MSL*). I define my variant of MSL to accomplish this in section A.1.

A design decision made here is not to model *higher-order functions*, i.e. functions taking functions as arguments again. Relatedly, functions are not "objects" in the sense of first-order logic: For example, one couldn't bind a variable to a function. Instead, the above "functions" will in fact be terms and "$\gg\!=$" does not actually occur as a symbol, but there will be one "$\gg\!=$"-symbol per term. This will be fleshed out in section A.1.3.

The reason for not allowing higher-order mechanisms lies in the models: While it is easy to come up with a MSL design where functions *are* first-class objects and a special evaluation function symbol is provided to apply a function (object) to an argument – or simply use a higher-order logic – it was seen in theorem 5.1 that there is no model of such a theory in the category $\mathcal{M}$ of measurable spaces which contains e.g. the real numbers.

Given that lambda notation is set up, one notices that Haskell code can be written to look similar to lambda notation[41] as long as certain restrictions with respect to higher-orderness are made. I will show in section A.2 how the correspondence can be made explicit to receive a translation from Haskell to MSL, i.e. a way to derive a formal specification in MSL from a Haskell program where the data types correspond to sorts, the functions correspond to function symbols and the models of the resulting MSL theory are meant reflect the intended semantics of the program. In particular, an abstract machine executing the program should be a model in the category of computable functions.

Finally, one can introduce the elementary data types and functions which constitute the theory $\mathrm{LPT}_{\mathrm{Prim}}$, the first part of LPT (section A.3). Some of these will be derived from Haskell code.

[41] This is not coincidental: Haskell implements a variant of the lambda calculus [5, sec. 1.2] and lambda notation tries to mimic some features of it. An introduction to the lambda calculus as well as a translation from the Haskell predecessor Miranda can be found in [14].

A.1 MSL and lambda notation

An introduction to many-sorted first-order logic (MSL) can be found in [15, chapter VI]. I make the following modifications to Manzano's approach:

- In Manzano's book, there is only one kind of symbol, namely functional symbols, forming the set `OPER.SYM`, and relations are functions to a special sort 0, which describes boolean values. Any formula is just an expression of type 0.

  I use a more traditional distinction between relational and functional symbols. There will be no special sort 0 and formulas will be different from expressions or terms. However, I *do* introduce a `Bool` type below (section A.3.1) as well functional symbols for *almost* all relational symbols (section A.1.4) to get back most of the behavior of Manzano's MSL.

- I provide a new meta-language layer called *functional terms / types* to apply functions by position instead of by variable name and to denote anonymous functions effectively (lambda notation). I borrow some notation, but not its expressive power, from the lambda calculus.

- Manzano's framework allows untyped relational symbols. I do not.

The following definitions provide my variant of MSL.

A.1.1 MSL

Assume that there is a totally ordered countably infinite set of *variables*.

Definition A.1 (Sort)**.** A set $\mathscr{S}$ of sorts is just some set. Typically, $\mathscr{S}$ is thought to consist of character strings.

Definition A.2 (Type)**.** Given a set $\mathscr{S}$ of sorts, a $\mathscr{S}$*-type* is of one of the following forms:

1. A *value type* is just a sort.
2. A *functional type* is of form either
   - $s \in \mathscr{S}$ a sort (such a functional type is called *trivial* or *constant*) or
   - $s \to \alpha$ where $s \in \mathscr{S}$ is a sort and α is a functional $\mathscr{S}$-type. s is then called the *argument type* and α the *result type*.

   Functional types typically form chains like $s_1 \to (s_2 \to (\ldots \to s))$ with several argument types and a final result (value) type. In this case, I leave out parentheses and write just $s_1 \to s_2 \to \ldots \to s$.
3. A *relational type* is of form $\mathscr{R}(s_1, \ldots, s_n)$ where $s_1, \ldots, s_n \in \mathscr{S}$ are sorts.

Note that the argument of a functional type cannot be of form $s' \to t'$ again, i.e. higher-order types are not supported, as previously mentioned. Functional types can be seen as an additional layer on top of the logic framework. They are "shallow" in that complete formulas, proofs etc. will not contain any trace of them.

Definition A.3 (Signature)**.** A *signature* Σ is a pair $\Sigma = (\mathscr{S}, \Gamma)$ together with a function $\mathscr{T}_\Gamma$ mapping the elements of Γ to functional or relational $\mathscr{S}$-types. Γ is called the set of *symbols*. I also write $f :: \alpha$ ("f has type α") for $\mathscr{T}_\Gamma(f) = \alpha$ and then $\mathscr{T}_\Gamma$ is usually given implicitly.

Definition A.4 (Value Term)**.** Given a signature $\Sigma = (\mathscr{S}, \Gamma)$ and $s \in \mathscr{S}$, a (value) Σ-*term* of (value) *type* s is of one of the following forms:

1. $x :: s$ where x is a variable.
2. $(f\ t_1\ \ldots\ t_n)$ where $t_1, \ldots, t_n$ are (value) terms of type $s_1, \ldots, s_n$, respectively, and $f \in \Gamma$ is a symbol of functional type $s_1 \to \ldots \to s_n \to s$.

Leave out parentheses if they are not required.

I write $t :: s$ when a (value) term t has (value) type s. I leave out sort specifies for variables if they are clear from the context.

Remark A.5. The previous definition technically allows the same variable to be used several times with different sort specifiers. For example, "$x ::$ `Int`" and "$x ::$ `Double`" would be simply be seen as different symbols. For obvious reasons, I never do this.

Note again how variables cannot be bound to (non-trivial) functional or relational types.

Definition A.6 (Functional Term)**.** Given a signature Σ, a Σ-*functional term* f of functional *type* α is of one of the following forms:

1. If α is a value type: f is a value term of type α.
2. If $\alpha = s \to \beta$: $f = \lambda x :: s.\ g$ where x is a variable and g is a functional term of type β.

As usual, I write $f :: \alpha$ to state that a f has functional type α.

Notation A.7. Write short

$$\lambda x_1\ \ldots\ x_n.\ g$$

for

$$\lambda x_1.\ (\lambda x_2.\ \ldots (\lambda x_n.\ g) \ldots)$$

and leave out parentheses by having extend lambda expressions as far to the right as possible, making the above equivalent to

$$\lambda x_1.\ \lambda x_2.\ \ldots \lambda x_n.\ g.$$

If $f \in \Gamma$ is a functional symbol of type $s_1 \to \ldots \to s_n \to s$, write just f for

$$\lambda x_1\ \ldots\ x_n.\ f\ x_1\ \ldots\ x_n,$$

thus interpreting functional symbols as functional terms.

Definition A.8 (Application of terms)**.** If s is a sort, α is a functional type, $f = \lambda x :: s.\ g :: s \to \alpha$ is a functional term and $t :: s$ is a value term, the *application* $f\ t$ of f to t is the functional term of type α resulting from f as follows:

- If $x :: s$ is parameter still in g, $f\ t = g$.
- Otherwise ($x :: s$ is context in g or does not occur), let $f\ t$ arise from replacing any occurrence of $x :: s$ in g by t.

Write $t\ t'\ t''$ for $(t\ t')\ t''$.

Remark A.9.

1. A certain similarity to (simply typed) lambda calculus can be seen here. However, since variables cannot have functional type, none of the more complex constructions (like e.g. non-terminating expressions) can be done.

2. In fact, my lambda expressions can be seen as just a value term together with a list of parameter variables: Any functional term f is of form $f = \lambda\, x_1\ \ldots\ x_n.\ t$ where t is a value term. I call $x_1, \ldots, x_n$ the *parameters* or *arguments* of f and the other variables the *context*.

3. The notation of function application by juxtaposition ($f\ t_1\ \ldots\ t_n$ instead of $f(t_1, \ldots, t_n)$) as well as the structure of functional types ($s_1 \to \ldots \to s_n \to s$ instead of $s_1 \times\ \ldots \times s_n \to s$) are borrowed from Haskell.

   Haskell and my notation also share the property that in fact, every function has only one argument, and applying an argument to a function yields a new function with one parameter less, where the parameter already applied would be stored in a closure in Haskell. This is called *partial application*. In terms of sets, it means that the two functions

   $$\begin{aligned} f &:: A \times B \to C \\ f(x,y) &= g(x)(y) \\ g &:: A \to C^B \\ g(x) &= (y \mapsto f(x,y)) \end{aligned}$$

   are identified. This identification is called *currying* and g is also called `curry` f and f is also called `uncurry` g.

   The concept of currying becomes relevant in the context of higher-order functions. Cf. section A.1.3 for how these are handled and section 5.2.2 for a nontrivial case.

4. The whole machinery is a second layer on top of the normal many-sorted first-order logic that helps specifying functions. As soon as formulas are concerned, functional terms are not mentioned any more. Relatedly, the arguments to function- or relational *symbols* of the language are always value terms.

   Allowing functional terms here could be a simple way to extend the framework to higher-order mechanisms, but this is intentionally not done here.

5. Relatedly, note how the argument of a functional term cannot be itself functional. The replacement would not even make sense because it would require a variable used as a function to stay syntactically valid. But this is not possible.

6. It should be mentioned that a variable bound by a lambda does not *have* to be used inside the defining value term. This provides a way to denote functional terms constant in a parameter.

   Note also that a functional term may specify the same parameter more than once. Then the term is constant in all but the last occurrence.

Example A.10. Given the machinery above, now the following is meaningful:

Given sorts `Int` and `Double` and symbols $(-)_{\texttt{Int}} :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}$, $(-)_{\texttt{Double}} :: \texttt{Double} \to \texttt{Double} \to \texttt{Double}$, $\texttt{floor} :: \texttt{Double} \to \texttt{Int}$ and $\texttt{asDouble} :: \texttt{Int} \to \texttt{Double}$, the following are functional terms (partly using my short notation):

- $t_1 := \lambda\, x\ y.\ (-)_{\texttt{Int}}\ (x :: \texttt{Int})\ (y :: \texttt{Int})$
- $t_2 := \lambda\, (y :: \texttt{Double})\ (x :: \texttt{Double}).\ (-)_{\texttt{Double}}\ x\ y$
- $t_3 := \lambda\, x.\ (-)_{\texttt{Double}}\ (\texttt{asDouble}\ (\texttt{floor}\ x))\ x$
- $t_4 := \lambda\, x.\ (+)_{\texttt{Double}}\ (t_3\ x)\ x$

The types are $t_1 :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}$, $t_2 :: \texttt{Double} \to \texttt{Double} \to \texttt{Double}$, $t_3 :: \texttt{Double} \to \texttt{Double}$ and $t_4 :: \texttt{Double} \to \texttt{Double}$.

From the names of the functions, one would expect that `floor` and `asDouble` come with axioms such that $t_4\ x = x$ for any x.

For $(-)_{\texttt{Double}}$ and $(-)_{\texttt{Int}}$ above, one would typically just write $(-)$. I do this from now on if the types are clear. Haskell provides a mechanism called *type classes* for this which are briefly mentioned in section A.2.5.

Notation A.11. When I write down lambda expressions, parameters are always applied explicitly. I assume that a parameter variable does not occur in the mentioned terms, except for at the place where the lambda is defined.

For example, if f is assumed to be a functional term of type $\mathbb{Z} \to \mathbb{Z}$ and I write "$g := \lambda\, x.\ f\ x \cdot 2$", then I assume that x does not occur as context in f. If f is e.g. $\lambda\, y.\ x + y$, then g should *not* be

$$\lambda\, x.\ (x + x) \cdot 2$$

where x being parameter in f and context in g leads to an unwanted name clash, but the parameter should be renamed to yield e.g.

$$\lambda\, z.\ (x + y) \cdot 2,$$

so the context is preserved.

As only parameters are renamed, no further modifications are necessary.

Definition A.12 (Formulas)**.** Given a signature $\Sigma = (\mathscr{S}, \Gamma)$, a Σ-*formula* ϕ takes one of the following forms:

- There are value terms t_1 and t_2 of the same result type and ϕ is of form $t_1 = t_2$.
- There is a relational symbol $R \in \Gamma$, $R :: \mathscr{R}\,(s_1, \ldots, s_n)$ where $s_1, \ldots, s_n \in \mathscr{S}$ are sorts and value terms $t_1, \ldots, t_n$ of result types $s_1, \ldots, s_n$, respectively, and $\phi = R\ t_1\ \ldots\ t_n$.

- ϕ is of form $\neg\psi$ or $(\psi \vee \psi')$ where ψ and ψ' are formulas.
- There is a formula ψ with a free variable $x :: s$ where $s \in \mathscr{S}$ is a sort and $\phi = \exists\, x :: s : \phi$.

A Σ-*theory* is a set of Σ-sentences (formulas without free variables). A *triple* is of form $(\mathscr{S}, \Gamma, \Phi)$ where $(\mathscr{S}, \Gamma)$ is a signature and Φ is a $(\mathscr{S}, \Gamma)$-theory.

As promised, the definition of formulas did not mention functional terms, so they can really be seen as a "convenience" layer on top of the usual logic framework.

Definition A.13 (Structures)**.** If $\Sigma = (\mathscr{S}, \Gamma)$ is a signature, a Σ-*structure* $\mathscr{A}$ consists of the following:

- A set $\mathrm{dom}\,(\mathscr{A})$.
- For any $s \in \mathscr{S}$ a subset $s^{\mathscr{A}} \subseteq \mathrm{dom}\,(\mathscr{A})$ such that $\bigcup_{s\in\mathscr{S}} s^{\mathscr{A}} = \mathrm{dom}\,(\mathscr{A})$.
- For $f \in \Gamma$ of functional type $s_1 \to \ldots \to s_n \to s$ a map $f^{\mathscr{A}} : {s_1}^{\mathscr{A}} \times \ldots \times {s_n}^{\mathscr{A}} \to s^{\mathscr{A}}$.
- For $R \in \Gamma$ of relational type $\mathscr{R}\,(s_1, \ldots, s_n)$ a subset $R^{\mathscr{A}} \subseteq {s_1}^{\mathscr{A}} \times \ldots \times {s_n}^{\mathscr{A}}$.

Manzano's MSL allows a hierarchy (i.e. a partial order) on the set of sorts to implement *subtyping*: A sort b may be marked "derived" from a sort a, allowing a variable of type b to be used in a a-context. While many object-oriented programming languages provide such a subtyping mechanism for their class hierarchy, Haskell does *not* do this, as does my translation below and the MSL variant here: While structures are not required to assign disjoint sets to different sorts, the subset relations would not be visible to the theory.

In Haskell, explicit conversion functions together with heavy use of type class polymorphism are applied to provide the features for which subtyping and implicit conversion are used elsewhere.[42] The MSL-translation will provide conversion mechanisms for certain cases where required.

Remark A.14. Let $\mathscr{A}$ be a Σ-structure.

If $f :: s_1 \to \ldots \to s_n \to s$ is a functional term with context types $t_1, \ldots, t_m$, in order, and $a_i \in {t_i}^{\mathscr{A}}$ for $i = 1, \ldots, m$, it is easy to define the interpretation

$$f^{\mathscr{A}}\,[\bar{a}] : {s_1}^{\mathscr{A}} \times \ldots \times {s_n}^{\mathscr{A}} \to a^{\mathscr{A}}$$

of f in $\mathscr{A}$ with context $\bar{a}$.

Given a Σ-structure $\mathscr{A}$, consider the category $\mathcal{C}_{\mathscr{A}}$ where the objects are of form $s^{\mathscr{A}}$ for $s \in \mathscr{S}$ (plus the cartesian products) and morphisms are of form $f^{\mathscr{A}}[\bar{a}]$ like above. Together with the interpretations of $\mathtt{id} = \lambda\, x.\ x$ and $f \circ g = \lambda\, x.\ f\ (g\ x)$, this indeed forms a category.

I call $\mathscr{A}$ a *structure in* a category $\mathcal{C}$ if $\mathcal{C}_{\mathscr{A}}$ can be enriched to a subcategory of $\mathcal{C}$. This can be made precise by requiring that there is a canonical forgetful functor from $\mathcal{C}$ to **Sets** and that $\mathcal{C}_{\mathscr{A}}$ is the image of a subcategory under that functor. The enrichment is typically mentioned together with the model $\mathscr{A}$. For example, for the most popular choice $\mathcal{C} = \mathcal{M}$, there are obviously several options, the trivial cases being the discrete or trivial σ-algebra for everything.

[42] Cf. the `from/toInteger/Rational` functions in the "Standard Prelude" [5, sec. 8].

It is left as an exercise to the reader to define what it means for a formula to hold in a structure and what a model of a theory is. No surprises are to be expected here. One receives proof rules analogous to sequent calculus where "types must match", and a variant of the completeness theorem. Translation to Manzano's MSL can be easily done as well.

A.1.2 Modification Operations

In the main part of this thesis, the following sections A.1.3 and A.1.4 and in section A.2 I construct, step by step, a triple as of definition A.12. To do this, it is required to add sorts, symbols, and axioms, and to iterate over all possible terms in some places. As adding symbols or sorts creates new terms, the process has to be repeated infinitely often.

The following definition provides a technical device to do this:

Definition A.15. A *modification operation* M is a map that constructs from a signature $(\mathscr{S}, \Gamma)$ a set $\mathscr{S}(M, \mathscr{S}, \Gamma)$ of new sorts, a set $\Gamma(M, \mathscr{S}, \Gamma)$ of new symbols and a set $\Phi(M, \mathscr{S}, \Gamma)$ of axioms.

If $\mathscr{M}$ is a set of modification operations, the *application* of $\mathscr{M}$ to a triple $(\mathscr{S}, \Gamma, \Phi)$ is the triple $(\mathscr{S}', \Gamma', \Phi')$ defined by induction on $\mathbb{N}$ as follows:

- $\mathscr{S}_0 = \mathscr{S}$, $\Gamma_0 = \Gamma$, $\Phi_0 = \Phi$
- $\mathscr{S}_{i+1} := \mathscr{S}_i \cup \bigcup_{M \in \mathscr{M}} \mathscr{S}(M, \mathscr{S}_i, \Gamma_i)$, analogous for Γ_{i+1} and Φ_{i+1}.
- $\mathscr{S}' := \bigcup_{i \in \mathbb{N}} \mathscr{S}_i$, analogous for Γ' and Φ'.

For $(\mathscr{S}', \Gamma', \Phi')$ to be in fact a triple again, $\mathscr{M}$ must be "complete enough" (any symbols/sorts mentioned in generated axioms must be added at some point). This will be the case for the modification operations used below.

Remark A.16. Note that a set of modification operations must be applied in parallel to receive the desired result: If $\mathscr{M} = \mathscr{K} \dot\cup \mathscr{L}$, then applying $\mathscr{M}$ to some triple is not the same as applying first $\mathscr{K}$ and then applying $\mathscr{L}$ to the result.

That's why modification operations should be thought of as being collected, then applied together to the empty triple. And so, a set of modification operations can be seen as a "generalized triple".

The following section A.2 as well as the sections 2 and 3 above can be thought of as defining certain modification operations when introducing new types and axioms. The lemmas stated in these sections hold in any triple that arises from application of the modification operations introduced until the respective lemma.

A.1.3 Closure Emulation

Consider a higher-order function like

$$\texttt{fmap} :: (a \to b) \to \texttt{Obs}\ a \to \texttt{Obs}\ b$$

from section 2. `fmap` applies a function "on top of" an observable.

Using `fmap`, one could then write e.g.

$$\begin{aligned} &\texttt{pluses} :: \ \texttt{Int} \to \texttt{Obs Int} \to \texttt{Obs Int} \\ &\texttt{pluses} := \lambda\, i\ io.\ \texttt{fmap}\ (\lambda\, j.\ i + j)\ io. \end{aligned}$$

$\texttt{pluses}\ i\ io$ adds i on top of the result of io.

Now there's a problem here, namely `fmap` being higher order: Its type $(a \to b) \to \texttt{Obs}\ a \to \texttt{Obs}\ b$ is not actually a valid (functional) MSL type as of above because the argument type $a \to b$ is not a value type. So `fmap` can't be a symbol. However, one can (intuitively) use `fmap` to define a function `pluses`, which is *not* higher-order, where `fmap` is applied to a function which is defined using the parameter i. In programming language terms, one would say that the parameter i is "stored in a closure".

To resolve the issue, one could quantify over all functional terms $f :: \texttt{Int} \to \texttt{Int}$ to receive many functional symbols $\texttt{fmap}_f :: \texttt{Obs Int} \to \texttt{Obs Int}$. But now there is no way to carry the closure parameter i!

The solution is to have the context i as an additional parameter to a $\texttt{fmap}_g$ symbol where g is $\lambda j.\ i + j$: We receive $\widetilde{\texttt{fmap}}_g :: \texttt{Int} \to \texttt{Obs Int} \to \texttt{Obs Int}$ and can write:

$$\begin{aligned}&\texttt{pluses} :: \ \texttt{Int} \to \texttt{Obs Int} \to \texttt{Obs Int}\\ &\texttt{pluses} := \lambda\, i\ io.\ \widetilde{\texttt{fmap}}_g\ i\ io\end{aligned}$$

Formally:

Definition A.17. Let f be a symbol and let α and β be functional types. The *closure emulation schema* $f :: \alpha \to \beta$ is the following modification operation:

For any functional term $g :: \alpha$ with context variables $y_1 :: b_1, \ldots, y_m :: b_m$, in order in the ordering of variables, add a new functional symbol

$$\widetilde{f}_g :: b_1 \to \ldots \to b_m \to \beta.$$

Then add the following axioms: Assume that $\alpha = a_1 \to \ldots \to a_k \to a$, $\beta = s_1 \to \ldots \to s_n \to s$ and let $g, h :: \alpha$. Let $\{y_1 :: b_1, \ldots, y_m :: b_m\}$ be the union of the context variables from g and h, in order. Add the following axiom:

$$\forall \bar{y} :: \bar{b} : (\forall \bar{z} :: \bar{a} : g\ \bar{z} = h\ \bar{z}) \to \left(\forall \bar{x} :: \bar{s} : \widetilde{f}_g\ \bar{y}\ \bar{x} = \widetilde{f}_h\ \bar{y}\ \bar{x}\right)$$

where "$\forall \bar{y} :: \bar{b}$" is short for "$\forall y_1 :: b_1 \ldots \forall y_m :: b_m$" and $g\ \bar{z}$ is short for $g\ z_1\ \ldots\ z_k$.

The axioms ensures that identical, with context, function arguments yield identical resulting functions.

f itself does not become a symbol and "$\alpha \to \beta$" is not actually a well-defined functional type, but we can use f *as if* it was:

Notation A.18. In the above situation, write $f\ g$ for the functional term

$$\begin{aligned}&\widetilde{f}_g\ \bar{y}\\ &= \lambda\, \bar{x} :: \bar{s}.\ \widetilde{f}_g\ \bar{y}\ \bar{x}.\end{aligned}$$

In this setting, I call the sorts $\bar{b}$ and the variables $\bar{y}$ the *closure context* of g in f.

Remark A.19. Note how …

1. the context of g has become context of $f\ g$, so the context is preserved.

2. application is done explicitly: $\widetilde{f_g}$ is just a symbol, but $\widetilde{f_g}\ \bar{y}$ is actual application of variables to a symbol. This is important in order to e.g. allow the proof calculus to rename variables.

3. the schema can easily be extended to higher-order functions with several functional parameters, which is not done here for simplicity.

4. only second-order functions are supported: The schema could not be applied to a function which takes a higher-order function as its argument. A generalization to arbitrary orders is possible, but not required here.

One can now write

$$\texttt{pluses} := \lambda\, i\ io.\ \texttt{fmap}\ (\lambda\, j.\ i + j)\ io$$

as wanted. Note again how j is not actually a variable in the RHS term: One only "sees" the symbol $\widetilde{\texttt{fmap}}_{\lambda\, j.\ i+j}$ and the variables i and io.

If the polymorphic version

$$\texttt{fmap} :: (a \to b) \to \texttt{Obs}\ a \to \texttt{Obs}\ b$$

is used, one even receives "polymorphic" (i.e. one for each choice of a and b) functions like

$$\begin{aligned}&\texttt{intoConst} :: \ a \to \texttt{Obs}\ b \to \texttt{Obs}\ a\\ &\texttt{intoConst} := \lambda\, x\ l.\ \texttt{fmap}\ (\texttt{const}\ x)\ l\end{aligned}$$

where $\texttt{const} = \lambda\, x\ y.\ x$, so $\texttt{const}\ x = \lambda\, y.\ x$.

Of course, we only added symbols so far. – In order for the symbols to have the intended behavior, one also needs to add axioms for them. Section A.2.3 only defines a translation for first-order functions, the axioms for higher-order functions are hand-crafted.

There are two use cases for higher-order functions in this thesis: The above-mentioned `fmap` from section 2 and the `case` functions from section A.2.2 below.

Remark A.20. Closure emulation is not a conservative extension: While a model might provide sensible interpretations for $m = 0$, it might fail to do so as soon as closures are involved. For example, in the model for LPT in the category $\mathcal{M}$ of measurable spaces, it must be explicitly shown that closure emulation for `fmap` is supported. Cf. section 5.2.2.

A.1.4 Lifted relations

In Manzano's MSL, there are not actually any "relational" symbols, just functions to the special two-element 0 type. The same is true for Haskell, of course. One might say that here, all relations are "computable" in a broad sense or *functionally lifted*.

However, this yields a semantic problem: In section 3, I introduce the `Con` type modeling financial contracts and a relation $\preceq :: \mathscr{R}\,(\texttt{Con}, \texttt{Con})$ indicating that a contract is "worth less than another one in present value". By the framework introduced in the sections 2 and 3, contracts may be defined using any kind of term and so the definitions of contracts would have access to this relation.

For separation of concerns, I decided that this should not be possible, and it can be avoided syntactically by making sure that "$\preceq$" is not a function. On the other hand, any other relation such as equality or "$\leq$" on numbers *should* be available for defining contracts, so these should be functions.

Definition A.21. The *functional lift* of a relational symbol $R :: \mathscr{R}(s_1, \ldots, s_n)$ (R may be equality) is the modification operation adding a new functional symbol $\hat{R}$:: $s_1 \to \ldots \to s_n \to \texttt{Bool}$ and the axiom

$$\forall \bar{x} :: \bar{s} : R(\bar{x}) \leftrightarrow \hat{R}\ (\bar{x}) = \texttt{True}.$$

R is then called *functionally lifted*.

All relations defined in this thesis will be functionally lifted, with the exception of "$\preceq$". All primitive types defined in section A.3 below will have functionally lifted equality, but equality on the types `Obs` a and `Con` will not be lifted.

Note that the other direction, namely converting from functions to formulas, is always possible: If $f :: s_1 \to \ldots \to s_n \to \texttt{Bool}$ is a term, then $(f\ \bar{x} = \texttt{True})$ is the corresponding quantifier-free formula.

A.2 Translating Haskell programs to MSL

With the general framework of MSL set up, one can now start to translate the different elements of the Haskell[5] language into MSL.

Haskell is a programming language very different from *imperative* languages like Java in that it has three properties making it particularly well-suited for formal considerations: Haskell is …

functional, meaning that functions are given as a sequence of applications of (possibly recursive) functions to terms rather than a series of commands.

pure, which means that functions never have side effects: All a function does is produce a value. It does not e.g. modify global variables. Aspects of a program which conceptually *are* effectful, such as I/O, are typically encapsulated in a concept called *monads*, which was also discussed in section 2.

strongly typed, which means that any expression has a type determined at compile time. Programs never[43] crash with a "type error" or "null pointer exception".

These three properties allow a Haskell function to be viewed as a mathematical formula, which is what I roll out in full detail in the following sections.

A Haskell program basically[44] consists of things:

1. Algebraic Data Type (ADT) definitions
2. Function definitions

[43] In practice, the type system is sometimes circumvented for reasons of convenience, but that would by no means be necessary.

[44] I leave out here things like the module system, type synonyms and other syntactic constructions which are rather a tool for the programmer than the core of the expressiveness of the language. – Any Haskell program could be written without these features.

3. Type class- and instance definitions
4. Every function has an associated type which needs to be translated as well.

For each of these four points, I will give a modification operation which translates them into MSL.

An important restriction is *higher-orderness*: In Haskell, functions are first-class objects just like any other piece of data: They can be stored in a variable and passed to other functions (so-called *higher-order functions*). Function types are ordinary types just like anything else. A function can create another function from its parameters. Such a function is then called a *closure*. As mentioned above, the MSL variant I am using follows a different approach. We will see that higher-order functions are still supported in some cases.

In addition to the above four points, I will introduce *abstract* data types like $\mathbb{R}$ which are not defined by their structure (like a ADT), but about which only certain properties are known. Haskell supports a similar mechanism also called "abstract data types": The internal representation of a type can be hidden through the module system. This happens e.g. for primitive types like `Double`.

In this section, many features of the Haskell language are ignored, examples being the module system, the foreign function interface, any syntactic construction such as named field in data types, almost any property of type classes as well as any advanced features such as generalized algebraic data types (GADTs) or type families which are typically implemented as language extensions. The result is a reduced subset of the language which is however sufficient to define all of the "standard" library as well as Peyton Jones' and Eber's framework and the framework introduced here.

A.2.1 Types

A *Haskell type* can take essentially[45] one of the following forms:

1. A *type variable*
2. Application of a *type constructor* (which is a name with a fixed arity) to a series of types or other type constructors. The *kind* of a type constructor defines which arguments are types and which are type constructors of which arity.

A special built-in type constructor is the function type constructor "`->`".

A function the type of which contains a type variable is called *polymorphic*: The type variables can be arbitrarily *instantiated*, i.e. replaced by types, and the function will still provide a definition for the more specific type. An example is

```
if' :: Bool -> a -> a -> a
if' True x y  = x
if' False x y = y
```

which expresses the "if" construct found in any programming language. A type without variables is called *ground* or *monomorphic*.

[45] Cf. [5, sec. 4.1.2]. As usual, I leave out more "convenience" features like tuples and special syntax for lists.

My translation to MSL does not preserve this structure:[46] A sort in MSL is just an arbitrary identifier. The MSL-types from above do *not* contain type variables. Hence, type variables need to be replaced by sorts for a Haskell type to translate to a MSL type. Polymorphism is instead treated on top of the meta language: A polymorphic function symbol is translated into many (monomorphic) function symbols, indexed by type instantiations. For example, the above example `if'` would become the set of symbols

$$\texttt{if'}_a :: \texttt{Bool} \to s \to s \to s$$

where s is a sort. The $\texttt{if'}_s$ are now proper MSL symbols. Of course, I will just write `if'` if the instantiation of a is arbitrary or clear from the context.

Special care has to be taken for Haskell's function type constructor "`->`": In Haskell, it can be mixed with other types, so a type like `Maybe (a -> b)` would be valid. In MSL, as functional types are not sorts, this is not possible. Hence, not all Haskell types are translatable. For simplicity, type constructors as arguments to other type constructors are not supported as well, but support for them could easily be added.

Write in the following short $\bar{s}$ for $s_1, \ldots, s_n$ etc.

Definition A.22. A Haskell type is called *functional* if it of form ν `->` ζ where ν and ζ are types.

A Haskell type is called *translatable* if arguments to type constructors are never other type constructors and functional types occur only at the top level or as the second (= RHS) argument of "`->`".

The *translation* of a translatable non-functional Haskell type ν with n type variables $u_1, \ldots, u_n$, in order, to MSL is the translation operation adding for any sorts $s_1, \ldots, s_n$ a new sort $\widetilde{\nu}(\bar{s})$ defined by replacing any occurrence of a type variable u_i in the string ν by the string s_i.

The *translation* of a translatable functional Haskell type ζ to MSL is a functional MSL-type defined likewise by inductively translating the contained non-functional translatable types to sorts, then replacing any instance of "`->`" by "$\to$". This will indeed be a functional MSL-type with respect to the sorts arising from the translation of the contained non-functional types.

If the type variables of a Haskell type η are contained in $\{u_1, \ldots, u_n\}$ (but are not necessarily exactly $u_1, \ldots, u_n$) and $s_1, \ldots, s_n$ are sorts, define the *named translation* $\widetilde{\eta}[\bar{u}/\bar{s}]$ to be the MSL type constructed just like above.

Note that the above produce two things: A translation operation and a resulting type $\widetilde{\eta}(\bar{s})$ for any choice of $\bar{s}$.

Remark A.23. The above translation operation does not recur: Translating a type like `Maybe Int` will just add this string as a sort, it does not automatically add a sort `Int` as well. However, any below translation will make sure that the contained types are added as well at some point.

Example A.24. Given the 0-ary Haskell type constructors `Int` and `Char`, unary type constructors `Maybe` and `List` and the binary type constructor `Either`, then

- the following are translatable non-functional Haskell types:
  `Int`, `List a`, `Either` (`List a`) (`Either b Int`).

[46] Of course, MSL, being a very general framework, *can* be used to give meaning to type variables and instantiation, but I chose a simple approach to typing here.

- the following are translatable functional Haskell types:
  `Int -> a`, `a -> Int`, `Either Int a -> b -> Char`
- the following are not translatable:
  `Maybe (a -> b)`, `(a -> b) -> c`

The type constructors `Maybe` and `List` are defined below.

The most popular GHC compiler provides a wide range of extensions to the type system [16, sec. 7] such as higher-rank types where quantifiers may be placed inside types. I do not support these and most extensions are interesting only together with higher-order functions. – which are only supported in a limited way:

A.2.2 Algebraic Data Types

Haskell supports[47], besides the built-in types of data such as IEEE floating point numbers (`Double`), algebraic data types (ADTs), i.e. a data type that is defined by a finite set of primitive functions (or *constructors*) . The constructors define the different (mutually exclusive) "shapes" a value of that type may have.[48] Functions can then *pattern match*, i.e. perform case distinction between the different "shapes". ADTs may be parameterized over other types. This structure is encoded into MSL as follows:

An Algebraic Data Type definition has a form as follows:

$$\texttt{data}\ T\ u_1\ \ldots\ u_k = K_1\ t_{1,1}\ \ldots\ t_{1,k_1} \mid \ldots \mid K_n\ t_{n,1}\ \ldots\ t_{n,k_n} \tag{A.1}$$

where `data` is a defined language keyword, $k \in \mathbb{N}$, $n \in \mathbb{N}$, $k_1, \ldots, k_n \in \mathbb{N}$, T and $K_1, \ldots, K_n$ are names, $u_1, \ldots, u_k$ are type variables and $t_{i,j}$ for $i \in \{1, \ldots, n\}$ and $j \in \{k_1, \ldots, k_n\}$ are Haskell types which may mention exactly the variables $u_1, \ldots, u_k$.[49][50] The $t_{i,j}$ may very well mention T or another ADT mentioning T again, leading to a *recursive ADT*.

Again, we need to restrict the possible values of $t_{i,j}$ to translatable non-functional types: A function cannot be stored in a data type in MSL while this is possible in Haskell.

Example A.25.

1. For $T =$ `Bool`, $k = 0$, $n = 2$, $K_1 =$ `True`, $K_2 =$ `False` and $k_1 = k_2 = 0$ one receives

   ```
   data Bool = True | False,
   ```

   i.e. the well-known type of boolean values: An object of type `Bool` can have one of exactly two possible abstract values, which are called `True` and `False`.

[47] Cf. [5, sec. 4.2.1]. I ignore features like record labels and strictness annotations as well as any language extensions.

[48] In fact, any ADT has an additional "shape" called "bottom", which models the computation that never terminates or an exception. I do *not* model "bottom". Non-termination will instead correspond to a inconsistent or underspecified theory. Cf. section A.2.3.

[49] Types and constructors are always set in upper case while anything else is set in lower case. This rule not only a convention, but part of the language. The language elements do not share a common namespace.

[50] Technically, $n = 0$ requires the `EmptyDataDecls` extension which is implemented e.g. in the GHC compiler [16, sec. 7.4.1].

2. Let $T = \texttt{Maybe}$, $k = 1$, $n = 2$, $K_1 = \texttt{Just}$, $k_1 = 1$, $t_{1,1} = u_1$, $K_2 = \texttt{Nothing}$ and $k_2 = 0$:

   ```
   data Maybe u1 = Just u1 | Nothing
   ```

   A value of type `Maybe` a is either of form `Just` x where x is of type a or of form `Nothing`. Thus, such a value is indeed "maybe an a". Note that this type is *parameterized*: It has $k = 1 > 0$ type parameter.

3. The canonical recursive ADT is `List`: Consider $T = \texttt{List}$, $k = 1$, $n = 2$, $K_1 = \texttt{Cons}$, $k_1 = 2$, $k_{1,1} = u_1$, $k_{1,2} = \texttt{List}\ u_1$, $K_2 = \texttt{Nil}$ and $k_2 = 0$:

   ```
   data List u1 = Cons u1 (List u1) | Nil
   ```

   A `List` a is either empty (form `Nil`) or it consists of a first element $x :: a$ and a remaining list $l :: \texttt{List}\ a$ (form `Cons` $x\ l$). x is sometimes called the *head* and l the *tail* of the list.

   Haskell offers special syntax for lists, writing $[a]$ for `List` a, $x : l$ for `Cons` $x\ l$ and $[x_1, \ldots, x_n]$ for `Cons` x_1 (`Cons` $\ldots$ (`Cons` x_n `Nil`)), but the definition is exactly equivalent.

   Lists do not need to be finite in Haskell, e.g. the list $[\texttt{0}, \texttt{1}, ..]$ of all natural numbers is easily definable. This is equally reflected in the translation: While not *enforcing* that infinite lists exist, the translation does not try to impose that lists be finite either.[51]

4. A tree type can be defined as follows:

   ```
   data Tree u1 = Branch u1 (List (Tree u1)) | Leaf
   ```

The following definition provides the translation in question:

Definition A.26. A ADT definition T as in equation (A.1) is called *translatable* if all the types $t_{i,j}$ are translatable and non-functional as of section A.2.1.

Given such a translatable ADT definition, the *translation* of T is the following modification operation:

1. Add a k-ary type constructor named T, i.e. perform the translation of the non-functional Haskell type $T\ u_1\ \ldots\ u_k$ from section A.2.1.

2. For $i = 1, \ldots, n$ and $s_1, \ldots, s_k$ sorts, add a functional symbol

   $$K_{i,\bar{s}} :: \widetilde{t_{i,1}}\,[\bar{u}/\bar{s}] \to \ldots \to \widetilde{t_{i,k_i}}\,[\bar{u}/\bar{s}] \to T\ \bar{s}$$

   where $\widetilde{t_{i,j}}[\bar{u}/\bar{s}]$ was defined in section A.2.1

   For any sorts s and $s_1, \ldots, s_k$, add symbols as of the closure emulation schema from section A.1.3 with respect to the higher-order function name

[51] An example where all ADT elements are finite expression in their constructors is the translation to (species and) measurable spaces in section D. An example with infinite lists is given in section A.2.6 below.

$\texttt{case}_{T,s,\bar{s}}$ and type

$$\begin{aligned}&\left(\widetilde{t_{1,1}}\,[\bar{u}/\bar{s}] \to \ldots \to \widetilde{t_{1,k_1}}\,[\bar{u}/\bar{s}] \to s\right)\\ \to\ &\ldots\\ \to\ &\left(\widetilde{t_{n,1}}\,[\bar{u}/\bar{s}] \to \ldots \to \widetilde{t_{n,k_n}}\,[\bar{u}/\bar{s}] \to s\right)\\ \to\ &T\ \bar{s}\\ \to\ &s\end{aligned}$$

This means the following: Whenever $f_1, \ldots, f_n$ are functional terms of types $f_i :: \widetilde{t_{i,1}}[\bar{u}/\bar{s}] \to \ldots \to \widetilde{t_{i,k_i}}[\bar{u}/\bar{s}] \to s$ such that the union of their context variables, in order, is $y_1 :: b_1, \ldots, y_m :: b_m$, add a new functional symbol

$$\widetilde{\texttt{case}}_{T,\bar{f}} :: b_1 \to \ldots \to b_m \to T\ \bar{s} \to s.$$

3. Add the following axiom for any choice of the sorts $\bar{s}$:

$$\forall y :: T\ \bar{s} : \dot{\bigvee_{i=1,\ldots,n}} \left(\exists \bar{x} :: \widetilde{\bar{t}_i}\,[\bar{u}/\bar{s}] : y = K_i\ \bar{x}\right) \tag{A.2}$$

where "$\exists \bar{x} :: \widetilde{\bar{t}_i}[\bar{u}/\bar{s}]$" is short for "$\exists x_1 :: \widetilde{t_{i,1}}[\bar{u}/\bar{s}] \ldots \exists x_{k_i} :: \widetilde{t_{i,k_i}}[\bar{u}/\bar{s}]$" and "$\dot{\bigvee}$" is short for the formula stating that "exactly one of them holds".

For any choice of $\bar{s}$, s and $\bar{f}$ in the definition of the `case` functions above and $i = 1, \ldots, n$, add the following axiom:

$$\forall \bar{y} :: \bar{b} : \forall \bar{x} :: \widetilde{\bar{t}_i}\,[\bar{u}/\bar{s}] : \texttt{case}_{T,\bar{f}}\ \bar{y}\ (K_i\ \bar{x}) = f_i\ \bar{x}\ \bar{y} \tag{A.3}$$

The first axiom from A.26.3 states that any ADT value must be defined by one of the ADT constructors K_i while the second states that one can use pattern matching to get the contained values back as arguments to functions.

Of course, I will leave out the indices whenever possible.

Example A.27.

1. The `Bool` ADT from above now indeed yields a sort `Bool` together with two 0-ary function symbols (i.e. constants) `True`, `False` :: `Bool` and the axiom

$$\forall y :: \texttt{Bool} : y = \texttt{True} \dot{\vee} y = \texttt{False}$$

as expected. One can now also define – using the closure emulation syntax from section A.1.3 – a function like

$$\begin{aligned}&\texttt{not} :: \ \texttt{Bool} \to \texttt{Bool}\\ &\texttt{not} := \texttt{case}_{\texttt{Bool}}\ \texttt{False}\ \texttt{True}\end{aligned}$$

and it follows from the axioms that this is a complete and consistent definition.

In practice, one would write the above as

$$\begin{aligned} \texttt{not True} &:= \texttt{False} \\ \texttt{not False} &:= \texttt{True}. \end{aligned}$$

Note how $\texttt{case}_{\texttt{Bool}}$ is usually called $\texttt{if}'$.

2. Likewise, for `Maybe`, one receives many new sorts $\texttt{Maybe}\ a$, functions $\texttt{Just}_a$ and constants $\texttt{Nothing}_a$ such that

$$\forall y :: \texttt{Maybe}\ a : y = \texttt{Nothing} \dot{\vee} \exists x :: a : y = \texttt{Just}\ x$$

and one can define functions (for any sort a)

$$\begin{aligned} \texttt{fromMaybe} &:: \ a \to \texttt{Maybe}\ a \to a \\ \texttt{fromMaybe} &:= \lambda\, x\ m.\ \ \texttt{case}_{\texttt{Maybe},a}\ \texttt{id}\ x\ m \end{aligned}$$

where $\texttt{id} = \lambda\, y.\ y$.

3. Finally, for `List`, one receives

$$\forall y :: \texttt{List}\ a : y = \texttt{Nil} \dot{\vee} \exists x :: a, l :: \texttt{List}\ a : y = \texttt{Cons}\ x\ l.$$

Note that it is *not* stated that $y \neq l$. And indeed, as soon as lists can be infinite – and there's no way to prevent that axiomatically – it is not clear whether this should hold. For example, Haskell allows a definition like this:

```
trues :: List Bool
trues = Cons True trues
```

Now it is not clear whether the tail of `trues` is truly *equal* to `trues` or just exhibits the same behavior as `trues`. The internals of the (GHC compiler's) Haskell runtime as well as the encoding from below section A.2.3 suggest that they should be equal.

One easily receives that constructors must be injective:

Lemma A.28. *Let be given a translation of a ADT definition as of definition A.26 and fix sorts $\bar{s}$. All the functions K_i are injective in the sense that for any i the following holds:*

$$\forall \bar{x}, \bar{y} :: \widetilde{\bar{t}_i} : K_i\ \bar{x} = K_i\ \bar{y} \to \bigwedge_{j=1,\ldots,k_i} x_j = y_j$$

Proof. Let $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, k_i\}$ and apply case distinction to the functions f_l, $l = 1, \ldots, n$, defined by

$$f_l = \begin{cases} \lambda\, x_1\ \ldots\ x_{k_i}\ \left(z :: \widetilde{t_{i,j}}\right).\ x_j & \text{if } l = i \\ \lambda\, x_1\ \ldots\ x_{k_i}\ \left(z :: \widetilde{t_{i,j}}\right).\ z & \text{if } l \neq i \end{cases}$$

where the variable z will occur as context in the `case` schema and is only required for well-definedness.

Now, if $\bar{x}, \bar{y} :: \widetilde{\bar{t}_i}$ are such that $K_i\ \bar{x} = K_i\ \bar{y}$ and $z :: \widetilde{t_{i,j}}$ is arbitrary, one receives

$$\begin{aligned} & x_j = f_i\ \bar{x}\ z \\ = & \widetilde{\texttt{case}}_{T,\bar{f}}\ (K_i\ \bar{x}) \\ = & \widetilde{\texttt{case}}_{T,\bar{f}}\ (K_i\ \bar{y}) \\ = & f_i\ \bar{y}\ z = y_j \end{aligned}$$ □

ADTs can be found in many places where a sense of "combining" or "case distinction" is required. For example, the set of functional types is actually a recursive non-parameterized ADT handled in the meta language, as is the set of formulas. Template Haskell [16, sec. 7.12] provides a mechanism to represent a Haskell program as a ADT during compilation. The category-theoretic constructions of the product and coproduct can be thought of as ADTs as well. I give a translation of general ADTs into the category of measurable spaces in section D.

A.2.3 Functions

A Haskell function definition is basically[52] of the form

```
f = e
```

where f is a name and e is a Haskell expression.

A Haskell expression is basically of one of the following forms:

1. A variable (which must be in scope).
2. A function name.
3. Application of a expression to another expression, denoted by juxtaposition.
4. A lambda term, of form `\`x `::` ν `->` e where x is a variable newly brought into scope, ν is a Haskell type and e is an expression.
5. A case distinction on an expression e' of a ADT type $T\nu_1\ \ldots\ \nu_k$ where $\nu_1, \ldots, \nu_k$ are Haskell types, being of form

   ```
   case e' of
     K1 x1 ... x_k1 -> e1
     ...
     Kn x1 ... x_kn -> en
   ```

   where $K_1, \ldots, K_n$ are the constructors of T, $x_1, \ldots, x_{k_i}$ are variables newly brought into scope for each i and $e_1, \ldots, e_n$ are Haskell expressions of the same type.

   The types of the variables x_i can be inferred from the definition of T and $\bar{\nu}$.

[52] Cf. [5, sec. 4.4.3]. As usual, Haskell supports many more syntactic features than listed here, e.g. pattern matching on the LHS of function definitions and many more, which can be easily translated into the form discussed here.

A Haskell function definition is then simply of form $f = e$ where f is a name and e is a (lambda) expression. Any Haskell expression has a type, but the compiler can usually infer it. In the following, I assume that the type of an expression is always given.[53] For the details of assigning a type to an expression, cf. [5, sec. 4.5].

A Haskell expression can be translated into a MSL term (for a certain signature) by recursively performing application of terms for function application (possibly with closure emulation), replacing "`\x ->`" by "$\lambda\, x.\ $" and replacing case distinctions by calls to the `case` functions from section A.2.2. This is formalized by the following definition.

Again, not every piece of Haskell code can be translated due to the restrictions of MSL with respect to higher-order functions.

Definition A.29. A Haskell expression e is called *translatable* if a variable is never of functional type and lambda expressions are not passed as arguments to functions.

Assume a 1:1 correspondence between Haskell's and MSL's variables and function names. Let e be a Haskell expression of Haskell type η and let $\bar{u}$ be the type variables occurring anywhere in the types of e and its sub-expressions, in order. Let $\bar{s}$ be sorts.

The *translation* of e, instantiated to $\bar{s}$, is a MSL term $\widetilde{e}[\bar{u}/\bar{s}]$ of type $\widetilde{\eta}[\bar{u}/\bar{s}]$ defined as follows: Write $\widetilde{e}$ for $\widetilde{e}[\bar{u}/\bar{s}]$ and $\widetilde{\eta}$ for $\widetilde{\eta}[\bar{u}/\bar{s}]$.

1. If $e = x :: \eta$ is a variable, then η is non-functional and $\widetilde{e} = x :: \widetilde{\eta}$ is just this variable.

2. If e is a function name and $\widetilde{\eta} = s_1 \to \ldots \to s_n \to s$, then $\widetilde{e} = e = \lambda\, x_1 :: s_1\ \ldots\ x_n :: s_n.\ e\ (x_1 :: s_1)\ \ldots\ (x_n :: s_n)$.

3. If $e = f\ g$ is application, then $\widetilde{e}$ is the term $\widetilde{f}\ \widetilde{g}$ received by application of terms as of section A.1.1.

4. If $e =$ `\x ::` ν `->` f is a lambda term, then ν is non-functional and $\widetilde{\eta} = \widetilde{\nu} \to \widetilde{\theta}$ where θ is the Haskell type of f and one can set $\widetilde{e} = \lambda\, x :: \widetilde{\nu}.\ \widetilde{f}$.

5. If e is a case distinction as above, let for $i = 1, \ldots, n$ $f_i = \widetilde{e_i}$ and translate $\widetilde{e} = \mathtt{case}_{T,\bar{f}}$.

Function definitions can be translated by just adding a new functional symbol and stating that it should equal its defining expression. This can in fact be done for any MSL term:

Definition A.30. If f is a new function name and ϕ is a MSL term of functional type $\alpha = s_1 \to \ldots \to s_n \to s$, then *adding a function* $f = \phi$ is the modification operation that adds a functional symbol $f :: \alpha$ and the axiom

$$\forall \bar{x} :: \bar{s} : f\ \bar{x} = \phi\ \bar{x}.$$

Recap that f is a plain symbol while $\phi\ \bar{x}$ is a value term received from ϕ by replacing variables.

[53] Defining Haskell functions this way requires at least two GHC extensions, namely `NoMonomorphismRestriction` (in order to have the compiler accept the lambda-style definition) and `ScopedTypeVariables` (in order to give all type annotations). Cf. [16, sec. 7].

If $f = e$ is a Haskell function definition of translatable functional Haskell type ζ with type variables $\bar{u}$, then *adding a Haskell function* is the modification operation adding, for any choice of sorts $\bar{s}$, a (MSL) function $f_{\bar{s}} = \widetilde{e}[\bar{u}/\bar{s}] :: \widetilde{\zeta}[\bar{u}/\bar{s}]$.

Notation A.31 (Function equality)**.** I abbreviate the above formula as

$$f = \phi.$$

A translated Haskell expression is a valid MSL-term with respect to the signature where all the functional symbols occurring in the term exist. Hence, if a function is *recursive*, i.e. its name occurs in its own definition, or is part of a recursive group of functions calling each other, this function must indeed be added as a symbol.[54] On the other hand, a non-recursive function definition can be seen as a shortcut for its defining expression, replacing any uses of the function by its definition.

Example A.32. Consider the Haskell definition of the `fromMaybe` function from above:

```
fromMaybe :: a -> Maybe a -> a
fromMaybe x (Just y) = y
fromMaybe x Nothing = x
```

This definition is equivalent to

```
fromMaybe = \x -> \m -> case m of
  Just y -> y
  Nothing -> x
```

which is translated by definition A.29 into the MSL terms, one per sort a,

$$\begin{aligned}&\texttt{fromMaybe} :: a \to \texttt{Maybe}\ a \to a\\ &\texttt{fromMaybe} = \lambda\, x\ m.\ \texttt{case}_{\texttt{Maybe}}\ (\lambda\, y.\ y)\ x\ m\end{aligned}$$

where the RHS uses the notation for closure emulation and is equal to

$$\texttt{case}_{\texttt{Maybe},(\lambda\, y.\ y),x}\ x\ m.$$

Here, the two functions $(\lambda\, y.\ y)$ and (x) (of trivial functional type) passed to $\texttt{case}_{\texttt{Maybe}}$ have in total one context parameter x which is then passed explicitly.

Remark A.33 (Effects of bottoms)*.* A Haskell function may "hangup" or "yield bottom", i.e. go into an infinite loop, on certain values. As in any Turing-complete language, this cannot be prohibited syntactically by the halting problem.

If a Haskell function that may "bottom" is translated into MSL, the result can be either inconsistent or underspecified. In the former case, models cannot give an interpretation for the inputs leading to "bottom", in the latter they are free to choose any interpretation. For example, consider the following code:

[54] Traditionally, the lambda calculus would provide a *recursion-* or *fixed-point combinator* Y that does recursion. Cf. [14, sec. 2.4.1]. One could add such a combinator here as well in a similar fashion to the closure emulation schema, but it would amount to essentially adding *any* functional term as a symbol.

```
f :: Int -> Int
f x = f x

g :: Int -> Int
g x = (g x) + 1
```

When run, both functions would go into an infinite loop for any input. However, the function f yields the axiom

$$\forall x :: \texttt{Int} : f\ x = f\ x,$$

which is true for any function, while g yields

$$\forall x :: \texttt{Int} : g\ x = (g\ x) + 1,$$

which implies $0 = 1$ and is hence false.

A.2.4 Adding higher-order functions

The above schema does not support higher-order functions, but at least support for second-order functions which are bound to function names can easily be added as follows. I consider only higher-order functions with a single functional argument, which is their first argument.

In definition A.29, allow as another case applications of form $e = f\ g$ where f is a name for a higher-order function of type $\alpha \to \beta$ and g has a Haskell type that translates to α and define $\widetilde{e}$ by the corresponding instance of the closure emulation schema.

If $f = e$ is a Haskell function definition of higher-order type like above, then e is of form $(z :: \xi)$ `->` e' where ξ is a functional Haskell type and e' is a Haskell expression of type – say – ζ that may use z like a function. Then do the following to add the function $f = e$ (for any instantiation of the type variables, which is kept implicit here):

- Execute the closure emulation schema $f :: \widetilde{\xi} \to \widetilde{\zeta}$.
- If $g :: \widetilde{\xi}$ is a functional MSL-term, let $\widetilde{e'}_g$ be the functional MSL-term resulting from first translating e' where z is treated like a functional symbol of type $\widetilde{\xi}$, then replacing z by g and performing all applications. Add the axiom
$$f_g = \widetilde{e'}_g.$$

Note how $\widetilde{e'}_g$ may refer to an instantiation of f again (f_g or some other instance), so higher-order functions may well be recursive.

A.2.5 Type Classes

Haskell provides a mechanism to group a set of types supporting certain operations into hierarchical *classes* [5, sec. 4.1]. This concept should not be confused with the idiom of a "class" from object-oriented programming: The term "type sets" might be more appropriate. The GHC compiler implements several extensions to the type class mechanism, such as multi-parameter type classes where combinations of more than one type may be grouped into classes [16, sec. 7].

For example, consider a class like this (from the Prelude, [5, sec. 8]):

```
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  ...
```

A type of class `Num` must be a member of the `Eq` and `Show` classes and provide a binary operation (+). Then the "+" symbol can be used for this type.

A polymorphic Haskell type may contain restrictions that the type variables must belong to certain classes. This is called *parametric polymorphism*.

An *instance declaration* adds a type to a class, providing the required operations. For example, one could define

```
instance Num Fraction where
  (+) (Frac d1 n1) (Frac d2 n2) =
    Frac (d1 * n2 + d2 * n1) (n1 * n2).
```

I do *not* support parametric polymorphism though the above approach could be easily modified to do so by remembering subsets of sorts. Instead, parametric polymorphism is handled directly on top of the meta language, i.e. I add sort-indexed symbols as required. For the above example of "+", I will below add symbols $(+)_{\mathbb{R}}$ and $(+)_{\mathbb{Z}}$, but write just (+) if the types are clear.

A.2.6 Effects of functions on models

A defined function relates the choice of models for the types it is defined on.

Example A.34. Consider the triple corresponding to the following Haskell code:

```
data List a = Cons a (List a) | Nil

length :: List a -> IntPlus
length Nil = 0
length (Cons x l) = (length l) + 1
```

Further require a "+" operation and constants $0, 1$ on `IntPlus`.

Let $\mathscr{A}$ be a model of this triple such that $\texttt{IntPlus}^{\mathscr{A}} = \mathbb{N}$, and the interpretations of $0, 1, (+)$ are as expected. Note that this cannot be axiomatically enforced by similar reasons as the well-known first-order indefinability of the natural numbers.

Then all lists are finite, i.e. for any sort s and any $l \in \texttt{List}\ s^{\mathscr{A}}$, there are $1_n, \ldots, 1_{\in} s^{\mathscr{A}}$ such that

$$l = \texttt{Cons}^{\mathscr{A}}\ x_1\ \big(\ldots\big(\texttt{Cons}^{\mathscr{A}}\ x_n\ \texttt{Nil}\big)\big)$$

and $n = \texttt{length}_a{}^{\mathscr{A}}\ l$.

Proof. Induction on the $\texttt{length}^{\mathscr{A}}$ of a list. The only value of length 0 is `Nil`, so the statement is trivial here. If l is a list of length $n+1$ and the statement holds for n, l must be of form $l = \texttt{Cons}^{\mathscr{A}}\ x\ l'$ (as the other case, `Nil`, has length 0). And $n+1 = \texttt{length}^{\mathscr{A}}\ l = \texttt{length}^{\mathscr{A}}\ l' + 1$, so $\texttt{length}^{\mathscr{A}}\ l' = n$. By the induction hypothesis, there are $x'_1, \ldots, x'_n$ for l' as above. Then setting

$$x_i = \begin{cases} x & \text{if } i = 1 \\ x'_{i-1} & \text{if } i = 2, \ldots, n+1 \end{cases}$$

yields the statement for l. □

Example A.35. Consider the triple corresponding to the Haskell code from example A.34 where `IntPlus` is replaced by `Int`.

There is a model $\mathscr{A}$ where $\texttt{Int}^{\mathscr{A}} = \mathbb{Z}$ and there are infinite lists (i.e. lists which are not finite in the notion above) and lists of negative length.

Proof. Define $\texttt{List}\ a^{\mathscr{A}}$ to be the set of pairs $(\overrightarrow{x}, i)$ where $\overrightarrow{x}$ is a finite or infinite sequence in $a^{\mathscr{A}}$ and $i \in \mathbb{Z}$ and if $\overrightarrow{x}$ is finite, then i is the length of $\overrightarrow{x}$. Define further

$$\begin{aligned}
\texttt{Nil}^{\mathscr{A}} &= (\emptyset, 0) \\
\texttt{Cons}^{\mathscr{A}}\ x\ (\overrightarrow{x}, i) &= (x\overrightarrow{x}, i+1) \\
\texttt{case}_{\texttt{List}\ a}{}^{\mathscr{A}}\ f_N\ f_C\ (\overrightarrow{x}, i) &= \begin{cases} f_N & \text{if } \overrightarrow{x} = \emptyset \\ f_C\ x\ \left(\overrightarrow{x'}, i-1\right) & \text{if } \overrightarrow{x} = x\overrightarrow{x'} \end{cases} \\
\texttt{length}^{\mathscr{A}}\ (\overrightarrow{x}, i) &= i.
\end{aligned}$$

It is clear that the axioms arising from the definition of `length` are fulfilled. For the axioms for the ADT `List`, note that if $(\overrightarrow{x}, i) \in (\texttt{List}\ a)^{\mathscr{A}}$, then $\overrightarrow{x} = \emptyset \Leftrightarrow (\overrightarrow{x}, i) = \texttt{Nil}$. From that, one receives that a list is of form `Nil` or `Cons` and that the `case` function is correct with respect to the axioms. Hence, this is a model.

A list which is both infinite and of negative length is $(\overrightarrow{x}, i)$ where $\overrightarrow{x}$ is infinite and $i < 0$. □

Remark A.36. The previous (pathological) example could be eliminated by introducing a new axiom that allows induction on ADTs: As `Nil` has non-negative length and if l has non-negative length, then so has $\texttt{Cons}\ x\ l$, it should follow that any list has non-negative length.

Recap that the `length` function in Haskell has one more possible value, namely "bottom", which is attained on infinite lists, but the theory should be able to view lists like they are finite.

Such an axiom is a subject of future work. It should be chosen powerful enough to deal with complex cases such as mutually recursive data types and cases where in total potentially infinitely many types are involved such as the following:

```
data V a = VNil | VCons a (V (V a))
```

A.3 Common data types and functions

The triple $\mathrm{LPT}_{\mathrm{Prim}}$ is the triple resulting from the empty triple by executing the modification operations associated to the following paragraphs and adding functional lifts for all relational symbols including equality.

The resulting theory is the LPT version of Haskell's Standard Prelude [5, sec. 8]. Some functions below are in fact taken from there.

A.3.1 Well-known ADTs and functions

For any $n \in \mathbb{N}$, add a *tuple type*[55]

[55] Recap that types and constructors do not share a common namespace: The ADT name `Tn` and the constructor name `Tn` just happen to be the same string. This is a common pattern for ADTs with a single constructor.

```
data Tn a1 ... an = Tn a1 ... an.
```

I also write $(x_1, \ldots, x_n)$ for $T_n\ x_1\ \ldots\ x_n$ and leave out calls to the $\texttt{case}_{T_n}$ functions. For $n = 0$, one receives the *unit type* $() = T_0$ with exactly one possible value which is also denoted $() = T_0$.

Add the following ADTs:

```
data Bool = True | False
data Maybe a = Just a | Nothing
```

Add the functions corresponding to the following Haskell code:

```
id :: a -> a
id x = x

(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)

(&&) :: Bool -> Bool -> Bool
True && True = True
-- all remaining cases:
x && y = False

(||) :: Bool -> Bool -> Bool
False || False = False
x || y = True

not :: Bool -> Bool
not True = False
not False = True
```

I use the symbols $\circ$, $\wedge$ and $\vee$, $\neg$ for (.), (&&), (||) and `not`, respectively.

A.3.2 Numeric types

The following paragraphs add the numeric types `Real` $= \mathbb{R}$, `RealPlus` $= \mathbb{R}^+$ etc. For the operations, it is important that types match: For example, there is no sensible definition for division as of type `Real` $\to$ `Real` $\to$ `Real`, but only as `Real` $\to$ `RealNZ` $\to$ `Real` where `RealNZ` should be like `Real` without 0. As mentioned before, subset relations between the different numeric types are modeled explicitly.

- Add new sorts `Real`, `RealPlus`, `RealNZ`, `Int`, `Nat`. Let $\mathscr{S}_{\texttt{Num}}$ be the set of these sorts. Define $\texttt{Real}^{\mathscr{A}} = \mathbb{R}$, $\texttt{RealPlus}^{\mathscr{A}} = \mathbb{R}^+$, $\texttt{RealNZ}^{\mathscr{A}} = \mathbb{R} \setminus \{0\}$, $\texttt{Int}^{\mathscr{A}} = \mathbb{Z}$ and $\texttt{Nat}^{\mathscr{A}} = \mathbb{N}$.

- Add the following functional symbols:

$$\begin{aligned}
0_X &:: X && \text{for } X \in \{\texttt{Real}, \texttt{RealPlus}, \texttt{Int}, \texttt{Nat}\} \\
1_X &:: X && \text{for } X \in \mathscr{S}_{\texttt{Num}} \\
(+)_X &:: X \to X \to X && \text{for } X \in \mathscr{S}_{\texttt{Num}} \setminus \{\texttt{RealNZ}\} \\
(-)_X &:: X \to X && \text{for } X \in \{\texttt{Real}, \texttt{RealNZ}, \texttt{Int}\} \\
(\cdot)_X &:: X \to X \to X && \text{for } X \in \mathscr{S}_{\texttt{Num}} \\
(/) &:: \texttt{Real} \to \texttt{RealNZ} \to \texttt{Real} \\
(\,\cdot\,^{\cdot})_X &:: X \to X \to X && \text{for } X \in \mathscr{S}_{\texttt{Num}} \setminus \{\texttt{Int}\}
\end{aligned}$$

  Use the canonical interpretations in the structure $\mathscr{A}$. Leave out the subscripts if the types are clear.

- Add the following relational symbols:

$$(\leq)_X :: \mathscr{R}(X, X) \text{ for } X \in \mathscr{S}_{\texttt{Num}}$$

  Use the canonical interpretations in the structure $\mathscr{A}$. As usual, write $x < y$ for $x \leq y \wedge \neg x = y$.

- For $s, t \in \mathscr{S}_{\texttt{Num}}$, add a functional symbol (projection)

$$\pi_{s,t} :: s \to t$$

  and let

$$\pi_{s,t}{}^{\mathscr{A}}\left(x \in s^{\mathscr{A}}\right) := \begin{cases} x & \text{if } x \in t^{\mathscr{A}} \\ z & \text{otherwise, where } z \text{ is some fixed element of } t^{\mathscr{A}} \end{cases}$$

- Add the first-order many-sorted theory of the structure $\mathscr{A}$ as axioms.

The π functions above are my approach to emulate some sense of sub-typing in a simple way. One can now define, for example,

$$\begin{aligned}
[\cdot]^+ &:: \texttt{Real} \to \texttt{RealPlus} \\
[x]^+ &:= \texttt{if}'\ (x \geq 0)\ (\pi_{\texttt{Real},\texttt{RealPlus}}\ x)\ 0.
\end{aligned}$$

Finally, one can define the expected notation for numeric types:

Notation A.37. Write short $\mathbb{R}$ for `Real`, $\mathbb{R}^+$ for `RealPlus`, $\mathbb{R}^*$ for `RealNZ`, $\mathbb{Z}$ for `Int` and $\mathbb{N}$ for `Nat`.

Omit applications of the π functions if it is clear what is meant.

Remark A.38. Just taking the theory of $\mathscr{A}$ is an easy way to get all the (first-order) properties one needs. For example, one directly receives that all the $\pi_{s,t}$ where $s^{\mathscr{A}} \subseteq t^{\mathscr{A}}$ commute, that they are compatible with the operations and orderings and that $\pi_{t,s} \circ \pi_{s,t} = \texttt{id}$ whenever $s^{\mathscr{A}} \subseteq t^{\mathscr{A}}$.

However, other axioms, which might not be desired, are included as well. For example,

$$\exists x :: \mathbb{R} : \nexists p, q :: \mathbb{Z} : x = (\pi_{\mathbb{Z},\mathbb{R}}\ p)/(\pi_{\mathbb{Z},\mathbb{R}^*}\ q)$$

prohibits that $\mathbb{Q}$ can be chosen instead of $\mathbb{R}$ in a model.

In the following, I only use the very basic properties of the numeric types, essentially only that one can do computations. So one may replace the axioms defined here by hand-crafted ones that allow a wider range of models.

The set of numeric types could be extended as required.

A.3.3 Time

Add new sorts `Time` and `TimeDiff` and the following functional symbols:[56]

$$\begin{aligned}(+) &:: \texttt{TimeDiff} \to \texttt{TimeDiff} \to \texttt{TimeDiff}\\ 0 &:: \texttt{TimeDiff}\\ (\leq) &:: \mathscr{R}(\texttt{TimeDiff}, \texttt{TimeDiff})\\ (\leq) &:: \mathscr{R}(\texttt{Time}, \texttt{Time})\\ \texttt{timeOffset} &:: \texttt{Time} \to \texttt{TimeDiff} \to \texttt{Maybe Time}\end{aligned}$$

`timeOffset` has `Maybe` result type because time should be allowed to be finite, so not all `TimeDiff` values can be added to all `Time` values and still yield a valid `Time`. I also write $(+)$ for `timeOffset`, which is actually abuse of notation.

Add the following axioms:

- $(\texttt{TimeDiff}, (+), 0, (\leq))$ should form a linearly ordered commutative monoid, i.e. $(+)$ should be associative and commutative, 0 neutral with respect to $(+)$ and $(+)$ should be strictly monotonic.
- `Time` should be linearly ordered as well and `timeOffset` should be compatible with $(+), 0, (\leq)$ on `TimeDiff` in the following sense:
  - $t + 0 = \texttt{Just}\ t$
  - If $t + \Delta t = \texttt{Just}\ t'$ and $t' + \Delta t' = \texttt{Just}\ t''$, then $t + (\Delta t + \Delta t') = \texttt{Just}\ t''$.
  - If $t < t'$ and $t + \Delta t = \texttt{Just}\ s$ and $t' + \Delta t = \texttt{Just}\ s'$, then $s < s'$.
  - If $\Delta t < \Delta t'$ and $t + \Delta t = \texttt{Just}\ s$ and $t + \Delta t' = \texttt{Just}\ s'$, then $s < s'$.

Further add a functional symbol

$$\iota_{\texttt{Time},\mathbb{R}} :: \texttt{Time} \to \mathbb{R}$$

and require that $\iota_{\texttt{Time},\mathbb{R}}$ be strictly monotonic. As usual, I will leave out the $\iota_{\texttt{Time},\mathbb{R}}$ and treat `Time` values as elements of $\mathbb{R}$.

Example A.39. The canonical model $\mathscr{A}$ of the numeric types from above can be extended in a number of ways to support `Time` and `TimeDiff`:

1: Let `Time` be of form $\{1, \ldots, T\}$ for some $T \in \mathbb{N}$ and use $\texttt{TimeDiff} = \mathbb{Z}$. Use the obvious interpretations for $\iota_{\texttt{Time},\mathbb{R}}$, operators and $(\leq)$. Define

$$\texttt{timeOffset}^{\mathscr{A}}(t, \Delta t) = \begin{cases}\texttt{Just}\ (t + \Delta t) & \text{if } -t + 1 \leq \Delta t \leq T - t\\ \texttt{Nothing} & \text{otherwise.}\end{cases}$$

2: Similarly, one can choose `Time` and `TimeDiff` freely in $\mathbb{N}, \mathbb{Z}, \mathbb{R}^+, \mathbb{R}$.

3: Let $\texttt{Time}^{\mathscr{A}}$ be the ordinal $\omega \cdot \omega = \{n \cdot \omega + m \mid n, m < \omega\}$. Let $\texttt{TimeDiff}^{\mathscr{A}} = \mathbb{Z}$. Define

$$\texttt{timeOffset}^{\mathscr{A}}(n \cdot \omega + m, \Delta t) = \begin{cases}\texttt{Just}\ (n \cdot \omega + m + \Delta t) & \text{if } \Delta t \geq -m\\ \texttt{Nothing} & \text{otherwise}\end{cases}$$

[56] The approach of having separate types for `Time` and `TimeDiff` can be found in the `time` Haskell package [17].

and

$$\iota_{\texttt{Time},\mathbb{R}}{}^{\mathscr{A}}(n \cdot \omega + m) = 3n + 1 - \frac{1}{m+1}$$

This model views `Time` as an infinite series of "days", indexed by n, each of length 1 and 2 apart from each other and each consisting of infinitely many discrete time steps which will get closer and closer together as the day progresses. One can reach via `timeOffset` exactly the points in time of the same day. If one replaces 2 by 0 above, days follow immediately upon each other.

For the case where `Time` is an ordinal, a σ-algebra is required for a model as in section 5.1. One can use the Borel sets with respect to the order topology. In the above countable example $\omega \cdot \omega$, this is just the discrete σ-algebra.

Remark A.40. It is tempting to add a function `timeDiff` :: `Time` $\to$ `Time` $\to$ `TimeDiff` and an embedding $\iota_{\texttt{TimeDiff},\mathbb{R}}$ :: `TimeDiff` $\to \mathbb{R}$ together with certain compatibility conditions. However, either of them would break the last example above:

If one has `timeDiff`, it is not clear what e.g. the time difference $\omega - 1$ is supposed to be (it's not an ordinal!), so one would have to extend `TimeDiff` considerably and then, through `timeOffset`, also `Time`. If one has $\iota_{\texttt{TimeDiff},\mathbb{R}}$, one would probably require that all time steps have the same length $\iota(1)$. Then one couldn't embed $\omega \cdot \omega$.

What *can* be done to measure the time elapsed between two points t, t' :: `Time` is to just use $\iota_{\texttt{Time},\mathbb{R}}(t') - \iota_{\texttt{Time},\mathbb{R}}(t)$.

One may also add functions `seconds`, `minutes`, ... of type $\mathbb{N} \to$ `TimeDiff`, but I shall not need these.

B Some proofs of monadic lifting properties

This section gives the proof details for some of the lemmas from section 2.

Proof of lemma 2.2. Proof by induction on n. For $n = 0$, the statement is $f = \texttt{join}\ (\texttt{return}\ f)$, which is just axiom (*Mo4). So let $n > 0$ and assume that the statement holds for $n - 1$.

Define

$$g := \lambda x_1.\ \texttt{lift}_{n-1}\ (f\ x_1)\ o_2\ \ldots\ o_n.$$

Then

$$\begin{aligned}
&\texttt{join}\ (\texttt{lift}_n\ f\ o_1\ \ldots\ o_n)\\
=\ &\texttt{join}\ (o_1 \gg\!= g)\\
=\ &\texttt{join}\ (\texttt{join}\ (\texttt{fmap}\ g\ o_1))\\
=\ &\texttt{join}\ (\texttt{fmap}\ \texttt{join}\ (\texttt{fmap}\ g\ o_1))\\
=\ &\texttt{join}\ (\texttt{fmap}\ (\texttt{join} \circ g)\ o_1)\\
=\ &o_1 \gg\!= (\texttt{join} \circ g)
\end{aligned}$$

where the third equality is axiom (*Mo3), the fourth is (*Fu2) and the others are just definitions.

By the induction hypothesis, we have that

$$\texttt{join} \circ g = \lambda x_1.\ o_2 \gg\!= \lambda x_2.\ \ldots o_n \gg\!= \lambda x_n.\ f\ x_1\ x_2\ \ldots\ x_n$$

and so follows the claim. □

Proof of lemma 2.3. Induction on n. For $n = 0$, the statement reduces to

$$\texttt{lift}_{m+1}\ f\ (\texttt{return}\ g) = \texttt{lift}_m\ (f\ g)$$

This follows directly from the definition of $\texttt{lift}_{m+1}$:

$$\begin{aligned}
&\texttt{lift}_{m+1}\ f\ (\texttt{return}\ g)\ o_2\ \ldots\ o_m\\
&\quad = \texttt{return}\ g \gg\!= \lambda x.\ \texttt{lift}_m\ (f\ x)\ o_2\ \ldots\ o_m\\
&\quad = (\lambda x.\ \texttt{lift}_m\ (f\ x)\ o_2\ \ldots\ o_m)\ g\\
&\quad = \texttt{lift}_m\ (f\ g)\ o_2\ \ldots\ o_m
\end{aligned}$$

Now assume $n > 0$ and assume the statement to be proven for $n - 1$. By definition of the lift functions and the monad laws for "$\gg\!=$", we have

$$\begin{aligned}
&\texttt{lift}_{m+1}\ f\ (\texttt{lift}_n\ g\ p_1\ \ldots\ p_n)\ o_1\ \ldots\ o_m\\
&\quad = \texttt{lift}_n\ g\ p_1\ \ldots\ p_n \gg\!= \lambda x.\ \texttt{lift}_m\ (f\ x)\ o_1\ \ldots\ o_m\\
&\quad = (p_1 \gg\!= \lambda y.\ \texttt{lift}_{n-1}\ (g\ y)\ p_2\ \ldots\ p_n) \gg\!= \lambda x.\\
&\qquad\ \texttt{lift}_m\ (f\ x)\ o_1\ \ldots\ o_m\\
&\quad = p_1 \gg\!= \zeta
\end{aligned}$$

where

$$\begin{aligned}
\zeta\ y := {} & (\mathtt{lift}_{n-1}\ (g\ y)\ p_2\ \ldots\ p_n) \gg\!= \lambda x. \\
& \quad \mathtt{lift}_m\ (f\ x)\ o_1\ \ldots\ o_m \\
= {} & \mathtt{lift}_{m+1}\ f\ (\mathtt{lift}_{n-1}\ (g\ y)\ p_2\ \ldots\ p_n)\ o_1\ \ldots\ o_m \\
\overset{\text{(IH)}}{=} {} & \mathtt{lift}_{m+n-1}\ (f \circ_{n-1} (g\ y))\ p_2\ \ldots\ p_n\ o_1\ \ldots\ o_m \\
= {} & \mathtt{lift}_{m+n-1}\ ((f \circ_n g)\ y)\ p_2\ \ldots\ p_n\ o_1\ \ldots\ o_m
\end{aligned}$$

Thus, by definition, we receive

$$p_1 \gg\!= \zeta = \mathtt{lift}_{m+n}\ (f \circ_n g)\ p_1\ \ldots\ p_n\ o_1\ \ldots\ o_m \qquad \square$$

Proof of lemma 2.4. For $n = 0$ there is nothing to show. So let $n > 0$ and assume that the statement holds for $n - 1$. We have that

$$\begin{aligned}
& \mathtt{lift}_n\ f\ (\mathtt{return}\ x_1)\ \ldots\ (\mathtt{return}\ x_n) \\
= {} & \mathtt{return}\ x_1 \gg\!= \lambda x_1'.\ \mathtt{lift}_{n-1}\ (f\ x_1')\ (\mathtt{return}\ x_2)\ \ldots\ (\mathtt{return}\ x_n) \\
= {} & \mathtt{lift}_{n-1}\ (f\ x_1)\ (\mathtt{return}\ x_2)\ \ldots\ (\mathtt{return}\ x_n) \\
= {} & \mathtt{return}\ (f\ x_1\ \ldots\ x_n)
\end{aligned}$$

where the last equality follows by induction hypothesis, the second is a monad law and the first is the definition of $\mathtt{lift}_n\ f$. $\square$

Proof of lemma 2.6. Start with 1: Any permutation can be defined as a chain of permutations which swap consecutive elements. Hence, it suffices to consider these only. By the recursive definition of $\mathtt{lift}_n$, it suffices to consider only the permutation (1 2) swapping the first two elements. Now,

$$\begin{aligned}
& \mathtt{lift}_n\ f_\pi\ o_{\pi^{-1}(1)}\ \ldots\ o_{\pi^{-1}(n)} \\
& \quad = \mathtt{lift}_n\ f_{(1\ 2)}\ o_2\ o_1\ o_3\ \ldots\ o_n \\
& \quad = o_2 \gg\!= \lambda x_1.\ o_1 \gg\!= \lambda x_2.\ \mathtt{lift}_{n-2}\ \left(f_{(1\ 2)}\ x_1\ x_2\right)\ o_3\ \ldots\ o_n \\
& \quad = \mathtt{join}\ (\mathtt{lift}_2\ \phi\ o_2\ o_1)
\end{aligned}$$

where $\phi = \lambda x_1\ x_2.\ \mathtt{lift}_{n-2}\ (f_{(1\ 2)}\ x_1\ x_2)\ o_3\ \ldots\ o_n$. By Axiom (*Ob2), this is equal to

$$\mathtt{join}\ (\mathtt{lift}_2\ (\lambda x_1\ x_2.\ \phi\ x_2\ x_1)\ o_1\ o_2)$$

But $\lambda x_1\ x_2.\ \phi\ x_2\ x_1 = \phi_{(1\ 2)}$ is equal to

$$\lambda x_1\ x_2.\ \mathtt{lift}_{n-2}\ (f\ x_1\ x_2)\ o_3\ \ldots\ o_n.$$

Hence, by unrolling the definition of $\mathtt{lift}_n$ again, one receives equality to $\mathtt{lift}_n\ f\ o_1\ \ldots\ o_n$.

2: For $n = 1$, there is nothing to show. For $n \geq 2$, it follows easily by unrolling the definition of $\mathtt{lift}_n$: Proof by induction on n:

$$\begin{aligned}
\mathtt{lift}_n\ f\ o\ \ldots\ o = o \gg\!= \lambda x_1.\ \mathtt{lift}_{n-1}\ (f\ x_1)\ o\ \ldots\ o \\
= o \gg\!= \lambda x_1.\ \mathtt{fmap}\ (\lambda x.\ f\ x_1\ x\ \ldots\ x)\ o \\
= \mathtt{lift}_2\ (\lambda x_1\ x.\ f\ x_1\ x\ \ldots\ x)\ o\ o \\
= \mathtt{fmap}\ (\lambda x.\ f\ x\ \ldots\ x)\ o
\end{aligned}$$

where in the middle parts, "..." should mean $n-1$ repetitions. The middle equality follows from the induction hypothesis and the last one is axiom (*Ob2).

3: Proof by induction on n. For $n = 0$, there is nothing to show (as $\texttt{const}_0 = \texttt{id}$). So let $n \geq 1$, assume that the statement holds for $n-1$ and consider the definition of $\texttt{lift}_n\ (\texttt{const}_n\ x)$:

$$\begin{aligned}
&\texttt{lift}_n\ (\texttt{const}_n\ x)\ o_1\ \ldots\ o_n \\
&= o_1 \gg\!= \lambda x_1.\ \texttt{lift}_{n-1}\ (\texttt{const}_n\ x\ x_1)\ o_2\ \ldots\ o_n \\
&= o_1 \gg\!= \lambda x_1.\ \texttt{lift}_{n-1}\ (\texttt{const}_{n-1}\ x)\ o_2\ \ldots\ o_n \\
&= o_1 \gg\!= \texttt{const}\ (\texttt{lift}_{n-1}\ (\texttt{const}_{n-1}\ x)\ o_2\ \ldots\ o_n) \\
&= \texttt{lift}_{n-1}\ (\texttt{const}_{n-1}\ x)\ o_2\ \ldots\ o_n
\end{aligned}$$

where the last equality is (Ob3') and the others are simple transformations. Now the claim follows by the induction hypothesis. □

Proof of lemma 2.7. 1: For $n = 1$, we have by the monad laws

$$\begin{aligned}
&\texttt{join}\ (\texttt{return}\ (o_1 \gg\!= \lambda x_1.\ g\ x_1)) \\
&= \texttt{join}\ (\texttt{return}\ (o_1 \gg\!= g)) \\
&= o_1 \gg\!= g = \texttt{join}\ (\texttt{fmap}\ g\ o_1).
\end{aligned}$$

So let $n > 1$ and assume that the statement holds for $n-1$. Using lemma 2.6.1 we have that the LHS is equal to

$$\begin{aligned}
&\texttt{join}\ \big(\texttt{lift}_n\ g_{(1\ 2)}\ o_2\ o_1\ o_3\ \ldots\ o_n\big) \\
&\overset{(a)}{=} o_2 \gg\!= \lambda x_2.\ \texttt{join}\ \big(\texttt{lift}_{n-1}\ \big(g_{(1\ 2)}\ x_2\big)\ o_1\ o_3\ \ldots\ o_n\big) \\
&\overset{(IH)}{=} o_2 \gg\!= \lambda x_2.\ \texttt{join}\ \big(\texttt{lift}_{n-2}\ \big(\lambda x_3\ \ldots\ x_n. \\
&\qquad o_1 \gg\!= \lambda x_1.\ g_{(1\ 2)}\ x_2\ x_1\ x_3\ \ldots\ x_n\big)\ o_3\ \ldots\ o_n\big) \\
&= o_2 \gg\!= \lambda x_2.\ \texttt{join}\ \big(\texttt{lift}_{n-2}\ \big(\lambda x_3\ \ldots\ x_n. \\
&\qquad o_1 \gg\!= \lambda x_1.\ g\ x_1\ \ldots\ x_n\big)\ o_3\ \ldots\ o_n\big) \\
&\overset{(a)}{=} \texttt{join}\ \big(\texttt{lift}_{n-1}\ \big(\lambda x_2\ \ldots\ x_n.\ o_1 \gg\!= \lambda x_1.\ x_1\ \ldots\ x_n\big)\ o_2\ \ldots\ o_n\big)
\end{aligned}$$

as required. Here, equalities (a) follow by two applications of lemma 2.2.

2: For $n = 0$, the statement follows from axiom (*Mo4). So let $n > 0$ and assume that the statement holds for $n-1$. Let $p_i :: \texttt{Obs}\ (\texttt{Obs}\ a_i)$ for $i = 1, \ldots, n$ and write short $\bar{p}$ for $p_1\ \ldots\ p_n$ and $\bar{p}_2$ for $p_2\ \ldots\ p_n$. We need to show:

$$\texttt{lift}_n\ f\ (\texttt{join}\ p_1)\ \ldots\ (\texttt{join}\ p_n) = \texttt{join}\ (\texttt{lift}_n\ (\texttt{lift}_n\ f)\ \bar{p})$$

By definition and (Mo3'), the LHS is equal to $p_1 \gg\!= \zeta$ where

$$\begin{aligned}
\zeta\ o_1 &:= o_1 \gg\!= \lambda x_1.\ \texttt{lift}_{n-1}\ (f\ x_1)\ (\texttt{join}\ p_2)\ \ldots\ (\texttt{join}\ p_n) \\
&\overset{(IH)}{=} o_1 \gg\!= \lambda x_1.\ \texttt{join}\ (\texttt{lift}_{n-1}\ (\texttt{lift}_{n-1}\ (f\ x_1))\ \bar{p}_2).
\end{aligned}$$

Using (a) from part 1, the RHS is equal to $p_1 \gg\!= \xi$ where

$$\xi\ o_1 := \texttt{join}\ (\texttt{lift}_{n-1}\ (\texttt{lift}_n\ f\ o_1)\ \bar{p}_2).$$

I show that $\zeta = \xi$:

$$\begin{aligned}
\zeta\ o_1 &\overset{\text{(a)}}{=} \texttt{join}\ (\texttt{lift}_n\ (\lambda x_1.\ \texttt{lift}_{n-1}\ (f\ x_1))\ o_1\ \bar{p}_2) \\
&\overset{\text{(part 1)}}{=} \texttt{join}\ (\texttt{lift}_{n-1}\ (\lambda o_2\ \ldots\ o_n.\ o_1 \gg\!= \lambda x_1.\ \texttt{lift}_{n-1}\ (f\ x_1)\ o_2\ \ldots\ o_n)\ \bar{p}_2) \\
&\overset{\text{(def)}}{=} \texttt{join}\ (\texttt{lift}_{n-1}\ (\texttt{lift}_n\ f\ o_1)\ \bar{p}_2) = \xi\ o_1
\end{aligned}$$

3 follows from 2: Recap that $o_i \gg\!= f_i = \texttt{join}\ (\texttt{fmap}\ f_i\ o_i)$ and so by part 2, the LHS is equal to

$$\texttt{join}\ (\texttt{lift}_n\ (\texttt{lift}_n\ f)\ (\texttt{fmap}\ f_1\ o_1)\ \ldots\ (\texttt{fmap}\ f_n\ o_n)).$$

Via lift collapsing (lemma 2.3 / remark 2.5), applied to the outer $\texttt{lift}_n$, this is equal to $\texttt{join}\ (\texttt{lift}_n\ g\ o_1\ \ldots\ o_n)$ as required. □

C Some proofs about atomic measurable spaces

This section contains some proofs from section 5.2.1.

Proof of lemma 5.5. 1: There is an atom $K \ni x$ by definition of "atomic". If there is another one $K' \ni x$, then $x \in K \cap K' \neq \emptyset$, so by minimality $K = K \cap K' = K'$.

2: If $A \in \mathcal{A}$ and $K \in \mathcal{A}$ is an atom, then $K \cap B$ is by minimality either $\emptyset$ or K. Hence:

$$B = \bigcup_{\omega \in B} \{\omega\} = \bigcup_{\omega \in B} K_\omega$$

3: Clear. It suffices to see that the atoms form a partition of X, which is also clear.

4: If there is $B \in \mathcal{A}(Y)$ with $f(x) \in B$ and $f(y) \notin B$, then $x \in f^{-1}(B)$ and $y \notin f^{-1}(B)$, i.e. x and y can be distinguished by $f^{-1}(B) \in \mathcal{A}$, contradicting 1.

If there is $B \in \mathcal{A}(Y)$ with $\emptyset \subsetneq B \subsetneq f[K]$, then in particular B separates two points in $f[K]$, contradicting the first part of 3. □

Proof of lemma 5.7. 1: I show that the set

$$\{E \subseteq X \times Y \mid \forall (x,y) \in E : K_x \times K_y \subseteq E\}$$

is a σ-algebra containing all rectangles.

1. $\emptyset$ and $X \times Y$ trivially have the property.

2. Rectangles: $A \times B$ with $A \subseteq X$, $B \subseteq Y$ is easily seen to fulfill the property.

3. Complement: Let $E \subseteq X \times Y$ be with the property. Let $(x,y) \in (\Omega \times \Omega) \setminus E$ and assume that $K_x \times K_y \not\subseteq (\Omega \times \Omega) \setminus E$, i.e. that there are $(x',y') \in E \cap K_x \times K_y$.

   Then, as E has the property, also $K_{x'} \times K_{y'} \subseteq E$. But $(x,y) \in (K_{x'} \times K_{y'}) \setminus E$. Contradiction. Hence, $(\Omega \times \Omega) \setminus E$ has the property.

4. Countable union: Follows easily as the property is local.

2: Again, I show that the set

$$\{E \in \mathcal{A}(X \times Y) \mid E_K \text{ is measurable}\}$$

is a sub-σ-algebra of $\mathcal{A}(X \times Y)$ containing all rectangles and must hence be identical to it.

1. $\emptyset_K = \emptyset$, $(X \times Y)_K = Y$.

2. Rectangles: Let $A \subseteq X, B \subseteq Y$ measurable.

$$(A \times B)_K = \begin{cases} B & \text{if } K \subseteq A \\ \emptyset & \text{otherwise} \end{cases}$$

3. Complements: Let $E \subseteq X \times Y$ be with the property. Let $y \in Y$. By 1, $K \times K_y \supseteq K \times \{y\}$ is an atom in $X \times Y$. Together with measurability of E, one receives

$$\begin{aligned} y \in \Omega \setminus E_K &\Leftrightarrow K \times \{y\} \not\subseteq E \\ &\Leftrightarrow K \times K_y \not\subseteq E \\ &\Leftrightarrow K \times K_y \subseteq (\Omega \times \Omega) \setminus E \\ &\Leftrightarrow K \times \{y\} \subseteq (\Omega \times \Omega) \setminus E \\ &\Leftrightarrow y \in ((\Omega \times \Omega) \setminus E)_K. \end{aligned}$$

   So $((\Omega \times \Omega) \setminus E)_K = \Omega \setminus E_K$ which is measurable by the inductive assumption.

4. Countable union: It is easy to see that $(\bigcup_i E_i)_K = \bigcap_i (E_i)_K$.

For $x \in X$, it is easy to see that $E_x = E_{K_x}$ similar to the step for complements above:

$$y \in E_x \Leftrightarrow (x, y) \in E \Leftrightarrow K_x \times \{y\} \subseteq E \Leftrightarrow y \in E_{K_x}$$

□

Proof of corollary 5.8. Let $C \subseteq Z$ be measurable.

$$\begin{aligned} f(x, \cdot)^{-1}(C) &= \{y \mid f(x, y) \in C\} \\ &= \left\{y \mid (x, y) \in f^{-1}(C)\right\} \\ &= f^{-1}(C)_x. \end{aligned}$$

This set is measurable by measurability of f and lemma 5.7.2. □

Proof of corollary 5.10. 1: By admissibility of Ω, B is the countable union of its atoms. And so

$$E_B = E_{\bigcup_{\omega \in B} K_\omega} = \bigcap_{\omega \in B} E_{K_\omega}$$

is a countable intersection of – by lemma 5.7.2 – measurable sets.

3: It is easy to see that

$$\pi_1[E] = \Omega \setminus ((\Omega \times \Omega) \setminus E)_\Omega$$

which is measurable by 1.

2: For $a \in A$ and $\omega \in \Omega$, we have $(a, \omega) \in E \Leftrightarrow \{a\} \times K_\omega \subseteq E$ as that latter set is contained in the atom $K_a \times K_\omega$ by lemma 5.7.1. Hence, $A \times \{\omega\} \subseteq E \Leftrightarrow A \times K_\omega \subseteq E$. And so

$$E_A = \bigcup \{K \subseteq \Omega \text{ atom} \mid A \times K \subseteq E\}$$

which is, being a countable union, measurable.

4: Just like 3. □

D Building measurable spaces for ADTs via species

The intuitive idea for building a model of an ADT $T\ a$ (cf. section A.2.2) with respect to a measurable space X goes as follows:

1. Build the intuitive "smallest set-model" of T inductively as a set, leaving "holes" / "markers" / "labels" where data of type X would be put. In such a set, any element is a finite expression in the constructors. For example, if $T\ a = \texttt{List}\ a$, one would take the smallest set that contains `Nil` and is closed under `Cons`.
2. Put a copy of the "content" X at each of the "labels".
3. Be sure to keep track of which copy went where in order to define the required morphisms.

One would then expect that manipulations of the "shape" such as appending an element or tree rotations can be done in the measurable-space interpretation as well and that measurable maps, i.e. manipulations of the "content" X yield measurable maps again. The latter can be expressed categorically in that every ADT is expected to give rise to a functor on $\mathcal{M}$.

Fortunately, there already is a framework for the first and third step above called combinatorial *species*.[57]

Definition D.1. A *species* is a functor[58] from the category of finite sets and bijections into the category of finite sets and arbitrary maps. A *species morphism* is a natural transformation of such functors, i.e. just a morphism in the category of species.

Application of a species F to a finite set U is written $F[U]$ and application to a bijection $\tau : U \to V$ is written $F[\tau] : F[U] \to F[V]$.

Let **Spec** be the category of species and species morphisms.

Note that, by functoriality, the image of a species always consists of bijections, but species morphisms might employ non-bijections.

The idea of the definition is that a species should assign a set of n *labels* to a set of *structures* where each of the labels marks a position in the structure. Species morphisms then map a structure to another structure of a different species such that they commute with relabeling: They should only operate on the "shape", not on the labels.

Species support "sum" ($+$) and "product" ($\bullet$) operations which "distribute" the given set of labels among disjoint union and cartesian product, respectively. A "fixed-point operator" μ is also supported which allows defining recursive species. Together with the primitive species $\mathbf{1}$ (point species) and $\mathbf{X}$ (identity), these can model the structure of Haskell ADTs. Cf. [18].

I will only cover species with a single label set parameter here. These can encode Haskell ADTs with a single type parameter. The generalization to *multisort species* is straightforward.

[57] For a quick introduction into species in the context of Haskell ADTs cf. [18]. I only made the minor change of relaxing the target category to receive the required species morphisms.

[58] For the category-theoretic concepts of a functor and natural transformation cf. [7] again. [18] also provides a more detailed definition of species.

Species provide a much more general framework for describing finite data structures than ADTs and I will now give a construction of a measurable space for arbitrary species.

Definition D.2. Let X be a measurable space. If U is a finite set, define the product space X^U analogously to X^n for $n = |U|$. X^U is interpreted as the set of functions from U to X. An element of X^U is called a *generalized tuple*. $X^\emptyset$ consists of a single element $\emptyset$, the empty function.

Let F be a species and if U is a finite set, choose the discrete σ-algebra[59] on $F[U]$. Define the measurable space $\mathcal{E}_X F$ as

$$\mathcal{E}_X F := \widetilde{\mathcal{E}}_X F / \sim$$

where

$$\widetilde{\mathcal{E}}_X F := \bigcup_{\substack{U \subseteq \mathbb{N} \\ \text{finite}}} \left(F[U] \times X^U \right)$$

and $((s, \bar{x}) \in F[U] \times X^U) \sim ((s', \bar{x}') \in F[U'] \times X^{U'})$ if there is a bijection $\tau : U \to U'$ such that $(s', \bar{x}') = \tau(s, \bar{x}) := (F[\tau](s), \bar{x} \circ \tau^{-1})$.

The idea of the above definition is that, following [18], the species should define the "structure" while the "content" is given externally by a map from the sets of labels to X. The content map should adjust with a relabeling. This is what "$\sim$" is for: For example, "$\sim$" guarantees that a pair of – say – a list and a content assignment $((1\ 2\ 3), (x_1, x_2, x_3))$ is considered equal to $((1\ 3\ 2), (x_1, x_3, x_2)$.

In the previous definition, I allow arbitrary finite subsets of $\mathbb{N}$ as label sets. This way, e.g. projections can be represented by just restricting a generalized tuple to a subset (lemma D.10). One could also have used sets of form $\{1, \ldots, n\}$ together with some normalizing relabeling.

The following lemma will make some statements about the structure of $\mathcal{E}_X F$ and lifting properties. For that, define

$$\widetilde{\mathcal{E}}_{X,U} F := F[U] \times X^U \quad \text{for } U \subseteq \mathbb{N} \text{ finite.}$$

This set is measurable by definition. Let $p : \widetilde{\mathcal{E}}_X F \twoheadrightarrow \mathcal{E}_X F$ be the projection map onto equivalence classes and define for $A \subseteq \widetilde{\mathcal{E}}_X F$ measurable the union of *orbits* intersecting A by

$$\begin{aligned} \operatorname{Orb} A &:= \left\{ y \in \widetilde{\mathcal{E}}_X F \mid \exists x \in A : y \sim x \right\} \\ &= p^{-1}(p[A]). \end{aligned}$$

Lemma D.3.

1. *If $A \subseteq \widetilde{\mathcal{E}}_X F$ is measurable, then* $\operatorname{Orb} A$ *is measurable as well.*
2. *Images of measurable sets in $\widetilde{\mathcal{E}}_X F$ under p are measurable in $\mathcal{E}_X F$. In fact, the measurable sets in $\mathcal{E}_X F$ are* exactly *the images of measurable sets in $\widetilde{\mathcal{E}}_X F$ under p.*

[59] Any σ-algebra that makes the images of bijections $F[\tau]$ and any natural transformation measurable would do. One could even use $\mathcal{M}$ as the target category for **Spec** instead of finite sets as long as the natural transformations used below such as inclusion into a sum are supported.

3. *If $\alpha : \widetilde{\mathcal{E}}_X F \to Y$ is a measurable map such that $\alpha(x) = \alpha(y)$ whenever $x \sim y$, then α gives rise to a measurable map*

$$\begin{aligned} \alpha/{\sim} : &\quad \mathcal{E}_X F \to Y \\ (\alpha/{\sim})([z]) &:= \alpha(z). \end{aligned}$$

4. *If $\alpha : \widetilde{\mathcal{E}}_X F \to \widetilde{\mathcal{E}}_Y F$ is a measurable map such that $\alpha(x) \sim \alpha(y)$ whenever $x \sim y$, then α gives rise to a measurable map*

$$\begin{aligned} \alpha/{\sim} : &\quad \mathcal{E}_X F \to \mathcal{E}_Y F \\ (\alpha/{\sim})([z]) &:= [\alpha(z)]\,. \end{aligned}$$

Proof. 1: By definition of "$\sim$",

$$\operatorname{Orb} A = \bigcup_{\substack{U,V \subseteq \mathbb{N},\, |U|=|V| \\ \tau : U \to V \text{ bijection}}} \hat{\tau}\left[A \cap \widetilde{\mathcal{E}}_{X,U} F\right]$$

where

$$\begin{aligned} \hat{\tau} : &\quad \widetilde{\mathcal{E}}_{X,U} F \to \widetilde{\mathcal{E}}_{X,V} F \\ \hat{\tau}(s, \bar{x}) &:= \tau(s, \bar{x}) = \big(F[\tau](s), \bar{x} \circ \tau^{-1}\big). \end{aligned}$$

$\hat{\tau}$ is the product of two maps which are known to be measurable[60] and hence measurable. By considering $\hat{\tau^{-1}}$, also images under $\hat{\tau}$ of measurable sets are measurable. Now $\hat{\tau}[A \cap \widetilde{\mathcal{E}}_{X,U} F]$ is measurable for any choice of (U, V, τ) and the union above is countable.

2: If $A \subseteq \widetilde{\mathcal{E}}_X F$ is measurable, then $p^{-1}(p[A]) = \operatorname{Orb} A$, which is measurable in $\widetilde{\mathcal{E}}_X F$ by 1 and hence $p[A]$ is measurable in $\mathcal{E}_X F$. The last sentence is clear: p is surjective, so any measurable set B in $\mathcal{E}_X F$ is of form $B = p[p^{-1}(B)]$.

3: Well-definedness of $\alpha/{\sim}$ is equivalent to compatibility with "$\sim$". For measurability, let $B \subseteq Y$ be measurable.

$$\begin{aligned} (\alpha/{\sim})^{-1}(B) &= \{[z] \mid \alpha(z) \in B\} \\ &= p\left[\alpha^{-1}(B)\right] \end{aligned}$$

which is measurable by 2 and measurability of α.

4 is just 3 applied to $p \circ \alpha$. □

Remark D.4. Multisets separate orbits: If $\bar{x}$ and $\bar{y}$ do not contain the same elements with same multiplicities, then $(s, \bar{x}) \not\sim (t, \bar{y})\, \forall s, t$. If $\bar{x}$ is a generalized tuple, write $\pi\bar{x}$ for the multiset consisting of the elements of $\bar{x}$. If $\pi \subseteq X$ is a finite multiset, let

$$\widetilde{\mathcal{E}}_\pi F := \left\{(s, \bar{x}) \in \widetilde{\mathcal{E}}_X F \mid \pi\bar{x} = \pi\right\}$$

and let $\mathcal{E}_\pi F$ be its image under p. Then the $\widetilde{\mathcal{E}}_\pi F$ form a partition of $\widetilde{\mathcal{E}}_X F$ and the $\mathcal{E}_\pi F$ form a partition of $\mathcal{E}_X F$.

[60] Recap that $(\bar{x} \mapsto \bar{x} \circ \tau^{-1})$ is just reordering / relabeling of generalized tuple components.

It is also clear that cardinalities of the sets $U \subseteq \mathbb{N}$ separate orbits: The sets

$$\mathcal{E}_{X,n}F := p\left[\widetilde{\mathcal{E}}_{X,n}F\right]$$
$$\text{where } \widetilde{\mathcal{E}}_{X,n}F := \bigcup_{U:\,|U|=n} \widetilde{\mathcal{E}}_{X,U}F$$

for $n \in \mathbb{N}$ form a partition of $\mathcal{E}_X F$.

Example D.5.

1. Let $F = \mathbf{0}$. $F[U] = \emptyset$ for all U and hence $\widetilde{\mathcal{E}}_X F$ and $\mathcal{E}_X F$ are $\emptyset$.

2. Let $F = \mathbf{1}$. $F[\emptyset] = \{*\}$ is the singleton set containing some non-label and $F[U] = \emptyset$ for any $U \neq \emptyset$. Hence, $\widetilde{\mathcal{E}}_{X,\emptyset}F = \{*\} \times \{\emptyset\}$ and all the other $\widetilde{\mathcal{E}}_{X,U}F$ are $\emptyset$. So $\mathcal{E}_X F$ is a single point.

3. Let $F = \mathbf{X}$. By definition, $F[\{u\}] = \{u\}$ and the other $F[U]$ are $\emptyset$. Any two singleton sets are related by a bijection and the bijection trivially carries to the F-structures. Hence, $(\{u\}, (x)) \sim (\{v\}, (y))$ iff $x = y$. Altogether, $\mathcal{E}_X F = (\widetilde{\mathcal{E}}_{X,1}F)/\!\sim \cong X$. The isomorphism is received by applying lemma D.3.3 to the map $((\{u\}, (x)) \mapsto x)$.

4. It is easy to see that $\mathcal{E}_X(F + G) \cong \mathcal{E}_X F \,\dot\cup\, \mathcal{E}_X G$. The isomorphism can again be constructed via lemma D.3.3.

   $F + G$ is indeed the coproduct in the category **Spec**: One receives the expected inclusions (species morphisms) $\iota_1 : F \to F + G$ and $\iota_2 : G \to F + G$ and the required universal property.

5. Let $F = \mathbf{X}^2 = \mathbf{X} \bullet \mathbf{X}$. $F[U] = \emptyset$ unless U is a two-element set and $F[\{u_1, u_2\}] = \{(u_1, u_2), (u_2, u_1)\}$. The elements of $\widetilde{\mathcal{E}}_X F$ are then of form $((u_1, u_2), \bar{x})$ where $\bar{x}$ is a map $\{u_1, u_2\} \to X$.

   A "$\sim$"-equivalent element is obtained as

   $$((u_2, u_1), (u_1 \mapsto \bar{x}(u_2),\ u_2 \mapsto \bar{x}(u_1)),$$

   but recap that we allow relabellings to different index sets as well. One can obtain an isomorphism to X^2 in one of the following two equivalent ways:

   - Given $((u_1, u_2), \bar{x})$, the result is $(\bar{x}(u_1), \bar{x}(u_2))$.
   - Given $((u_1, u_2), \bar{x})$, find $\bar{y} \in X^2 = X^{\{1,2\}}$ such that $((1, 2), \bar{y}) \sim ((u_1, u_2), \bar{x})$. This exists and is unique. Then let the result be $\bar{y}$.

   I will show that general "$\bullet$"-products in **Spec** correspond to products in $\mathcal{M}$ in lemma D.10.

   Note that $F \bullet G$ is *not* the categorical product of F and G in **Spec**! To see that, try to construct a projection $\pi_1 : F \bullet G \to F$: This would have to be a natural transformation, so for any label set U one would need a map

   $$(F \bullet G)[U] = \dot{\bigcup_{U = U_1 \dot\cup U_2}} F[U_1] \times G[U_2] \ \to\ F[U].$$

However, such a map cannot be sensibly defined. For example, consider the above case of $F = G = \mathbf{X}$: For $|U| = 2$, there is no map $\pi_{1,U} : \emptyset \neq \mathbf{X}^2[U] \to \mathbf{X}[U] = \emptyset$.

The problem here is that $(F \bullet G)$ "distributes" the labels in U to both the F and G sides of the product while one would want to extract a structure that corresponds only to a subset of the labels. This can be done by mapping not to F, but to $\mathcal{S}F$ defined right below.

The product in **Spec** is in fact given by the cross product $F \times G$ that "applies F and G to the same label set at the same time".

In order to receive the projections out of "$\bullet$", one needs to consider a more general construction:[61]

Definition D.6. If F is a species, let $\mathcal{S}F$ be the species defined by

$$\mathcal{S}F[U] := \dot{\bigcup_{V \subseteq U}} F[V]$$
$$\mathcal{S}F[\tau : U \to U'](s \in F[V]) := F[\tau|_V](s) \in F[\tau\,[V]] \subseteq \mathcal{S}F[U'].$$

It is easy to see that $\mathcal{S}F$ is indeed a species, i.e. that it is functorial on relabellings.

Note that the union above is disjoint. I write $s \in F[V \subseteq U]$ to make clear that I mean s as an element of the V-component of $\mathcal{S}F[U]$.

Remark D.7.

1. It can be shown that $\mathcal{S}F \cong F \bullet \mathbf{E}$ where $\mathbf{E}$ is the species of sets mapping any set of labels to the singleton containing itself. Yorgey [18] in fact mentions briefly that "$\bullet\mathbf{E}$" can be used as a "sink" to mark labels optional. This is exactly what's happening here.

2. $\mathcal{S}$ is in fact a functor **Spec** $\to$ **Spec**: Given a species morphism $f : F \to G$, define $\mathcal{S}f : \mathcal{S}F \to \mathcal{S}G$ by $(\mathcal{S}f)_U(s \in F[V \subseteq U]) := f_V(s) \in G[V \subseteq U]$. It is easy to see that this mapping is indeed functorial.

3. It is further easy to see that $F \bullet \mathbf{E} \cong (\mathbf{X} \bullet \mathbf{E}) \circ F$ where "$\circ$" is species composition.[62] So $\mathcal{S}$ is given simply by the species $\mathbf{X} \bullet \mathbf{E}$. One can show that *any* transformation $(F \mapsto H \circ F)$ where H is some species gives rise to a functor.[63] Such a functor "adds a second layer of structure on top" of an existing species. For example, $F \mapsto \mathbf{B} \circ F$ replaces e.g. lists with trees of lists.

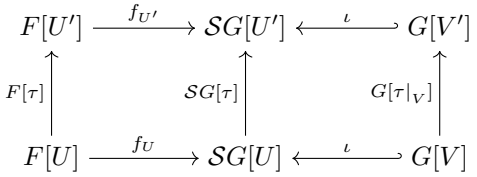
   In comparison, $F \mapsto F \circ \mathbf{B}$ would replace lists by lists of trees, which is obviously not the same: The new tree layer is added "below" the existing structure. Transformations defined by precomposition give rise to functors as well.

[61] We will see in section D.2 that this in fact means transition to another category.

[62] Recap from [18] that "$\circ$" is *not* functor composition! Rather, the available labels are partitioned and for each part, a F-structure is chosen. Then these structures are used as the label set for $\mathbf{X} \bullet \mathbf{E}$, which will essentially just pick one of them.

[63] For the morphisms, one needs to assume that no structure can be defined on two different label sets at the same time, which can always be ensured isomorphically.

Figure 12 Mapping in the species part of $\widetilde{\mathcal{E}}_X f$



Lemma D.8. *Let $f : F \to \mathcal{S}G$ be a species morphism and X a measurable space. Then f gives rise to a measurable map*

$$\begin{aligned} \mathcal{E}_X f : \quad & \mathcal{E}_X F \to \mathcal{E}_X G \\ \mathcal{E}_X f\Big(\Big[(s, \bar{x}) \in \widetilde{\mathcal{E}}_{X,U} F\Big]\Big) := & \Big[\Big(f_U(s), \bar{x}|_{V(f_U(s))}\Big)\Big] \end{aligned}$$

where $V(f_U(s)) \subseteq U$ is such that $f_U(s) \in G[V(f_U(s)) \subseteq U]$.

Proof. I apply lemma D.3.4 to

$$\begin{aligned} \widetilde{\mathcal{E}}_X f : \quad & \mathcal{E}_X F \to \mathcal{E}_X G \\ \widetilde{\mathcal{E}}_X f\Big((s, \bar{x}) \in \widetilde{\mathcal{E}}_{X,U} F\Big) := & (f_U(s), \bar{x}|_V) \end{aligned}$$

where $V := V(f_U(s))$.

Measurability: The space $\widetilde{\mathcal{E}}_X G$ is generated by sets $N \times B_{B,v,V}$ where $V \subseteq \mathbb{N}$ is finite, $N \subseteq G[V]$, $v \in V$ and

$$B_{B,v,V} := \big\{\bar{x} \in X^V \mid \bar{x}(v) \in B\big\}$$

is a generator of X^V. Consider the preimages of these sets under $\widetilde{\mathcal{E}}_X f$:

For some U and $(s, \bar{x}) \in \widetilde{\mathcal{E}}_{X,U} G$ we have $\widetilde{\mathcal{E}}_X f(s, \bar{x}) \in N \times B_{B,v,V}$ if $V \subseteq U$, $f_U(s) \in N \subseteq G[V \subseteq U]$ and $(\bar{x}|_V)(v) \in B$, i.e. if $s \in f_U^{-1}(N)$ and $\bar{x}(v) \in B$. Hence:

$$\Big(\widetilde{\mathcal{E}}_X f\Big)^{-1} (N \times B_{B,v,V}) = \bigcup_{\substack{U \subseteq \mathbb{N} \text{ finite} \\ U \supseteq V}} \big(f_U^{-1}(N) \times B_{B,v,U}\big)$$

This countable union is measurable by measurability of the maps f_U.

Compatibility with "$\sim$": Let $(s, \bar{x}) \in \widetilde{\mathcal{E}}_{X,U} F$, $(t, \bar{y}) \in \widetilde{\mathcal{E}}_{X,U'} F$ and let $\tau : U \to U'$ be a bijection relating the two, i.e. $\tau(s, \bar{x}) = (t, \bar{y})$. Let $V := V(f_U(s))$ and $V' = \tau[V]$.

By naturality of f and definition of $\mathcal{S}G$, diagram 12 commutes, so $G[\tau|_V](f_U(s)) = f_{U'}(t)$. Further, $\bar{x}|_V \circ (\tau|_V)^{-1} = (\bar{x} \circ \tau^{-1})|_{V'} = \bar{y}|_{V'}$. So $\tau|_V$ relates $\widetilde{\mathcal{E}}_X f(s, \bar{x})$ and $\widetilde{\mathcal{E}}_X f(t, \bar{y})$. □

Remark D.9. If F is a species, define the following species morphisms:

$$\begin{aligned} \texttt{return}_F : \quad & F \to \mathcal{S}F \\ \texttt{return}_{F,U}(s \in F[U]) := & \; s \in F[U \subseteq U] \subseteq \mathcal{S}F[U] \end{aligned}$$

`return` just maps a structure to itself in the "top layer" of $\mathcal{S}F$.

`return` can be used to lift "normal" species morphisms $f : F \to G$ to measurable functions $\mathcal{E}_X F \to \mathcal{E}_X G$ by lifting $\texttt{return} \circ f : F \to \mathcal{S}G$ instead. The resulting measurable function will then separately map $\mathcal{E}_\pi F$ to $\mathcal{E}_\pi G$ for any multiset π. I write just $\mathcal{E}_X f$ for $\mathcal{E}_X(\texttt{return} \circ f)$.

It is easy to see that any isomorphism $f : F \xrightarrow{\sim} G$ gives rise to a isomorphism $\mathcal{E}_X(\texttt{return} \circ f)\mathcal{E}_X F \xrightarrow{\sim} \mathcal{E}_X G$ with inverse $\mathcal{E}_X(\texttt{return} \circ f^{-1})$.

There is also an accompanying `join` turning $\mathcal{S}$ into a monad. This will give rise to a category $\mathcal{E}_X \cdot$ forms a functor on. Cf. section D.2.

One can now define the projections as species morphisms: Given F, G species, let

$$\begin{aligned}\pi_1 &: (F \bullet G) \to \mathcal{S}F \\ \pi_{1,U}((s,t) \in F[V] \times G[W]) &:= s \in F[V \subseteq U] \\ &\quad \text{where } V, W \text{ are such that } U = V \dot\cup W\end{aligned}$$

and π_2 analogous. It is easy to see that these commute with relabellings.

Lemma D.10. *Let F, G be species and X a measurable space.*

1. *The species morphisms $\iota_1 : F \to F + G$ and $\iota_2 : F \to F + G$ induce an isomorphism of measurable spaces*

$$\mathcal{E}_X F \dot\cup \mathcal{E}_Y G \xrightarrow{\sim} \mathcal{E}_X(F + G).$$

2. *The species morphisms $\pi_1 : F \bullet G \to \mathcal{S}F$ and $\pi_2 : F \bullet G \to \mathcal{S}G$ induce an isomorphism of measurable spaces*

$$\mathcal{E}_X(F \bullet G) \xrightarrow{\sim} \mathcal{E}_X F \times \mathcal{E}_X G.$$

Proof. 1: Recap that $(F + G)[U] = F[U] \dot\cup G[U]$ and $\iota_{1,2}$ are the inclusions. I identify $A \dot\cup G$ with $(A \times \{1\}) \cup (B \times \{2\})$.

Define $\xi := \mathcal{E}_X \iota_1 \dot\cup \mathcal{E}_X \iota_2$. We have

$$\begin{aligned}\xi :&\ \ \mathcal{E}_X F \dot\cup \mathcal{E}_X G \to \mathcal{E}_X(F + G) \\ \xi([(s, \bar x)], 1) &= [((s,1), \bar x)] \\ \xi([(t, \bar y, 2)]) &= [((t,2), \bar y)]\end{aligned}$$

where one should recap that $\mathcal{E}_X(F + G) = \bigcup_U ((F[U] \dot\cup G[U]) \times X^U)$. It is known from the previous discussion that ξ is well-defined and measurable.

I define the inverse map. Let

$$\begin{aligned}\widetilde\zeta :&\ \ \widetilde{\mathcal{E}}_X(F + G) \to \mathcal{E}_X F \dot\cup \mathcal{E}_X G \\ \widetilde\zeta((s,1), \bar x) &:= (([s], \bar x), 1) \\ \widetilde\zeta((t,2), \bar x) &:= (([t], \bar x), 2).\end{aligned}$$

$\widetilde\zeta$ is clearly measurable. If $((s,i), \bar x) \sim ((t,j), \bar y)$ then by definition of $F + G$ $i = j$ and $(s, \bar x) \sim (t, \bar y)$, so $\widetilde\zeta$ is compatible with "$\sim$". By lemma 3, one receives its factorization $\zeta := \widetilde\zeta/\!\sim : \mathcal{E}_X(F + G) \to \mathcal{E}_X F \dot\cup \mathcal{E}_X G$. It is easy to see that ζ and ξ are inverse.

2: Recap that $(F \bullet G)[U] = \bigcup_{V \dot\cup W = U} (F[V] \times G[W])$ and define $\xi := \mathcal{E}_X \pi_1 \times \mathcal{E}_X \pi_2$. We have

$$\xi : \quad \mathcal{E}_X(F \bullet G) \to \mathcal{E}_X F \times \mathcal{E}_X G$$
$$\xi\Big(\Big[((s,t), \bar{x}) \in (F[V] \times G[W]) \times X^{(V \dot\cup W)}\Big]\Big) = ([(s, \bar{x}|_V)], [(t, \bar{y}|_W)]).$$

For the inverse, define

$$\zeta : \quad \mathcal{E}_X F \times \mathcal{E}_X G \to \mathcal{E}_X(F \bullet G)$$
$$\zeta([(s,\bar{x})], [(t,\bar{y})]) := [((s,t), \bar{x} \cup \bar{y})] \qquad \text{if } \operatorname{dom}(\bar{x}) \cap \operatorname{dom}(\bar{y}) = \emptyset.$$

ζ is well-defined:

1. The constraint $\operatorname{dom}(\bar{x}) \cap \operatorname{dom}(\bar{y}) = \emptyset$ still includes all elements of $\mathcal{E}_X F \times \mathcal{E}_X G$: Whenever $(s, \bar{x}) \in \widetilde{\mathcal{E}}_{X,U} F$ and $U' = \operatorname{dom}(\bar{y})$ is some set, let U be a set of the same cardinality as U disjoint from U', pick a bijection $\tau : U \to U'$ and let $(s', \bar{x}') = \tau(s, \bar{x})$. Then $[(s', \bar{x}')] = [(s, \bar{x})]$ and $\operatorname{dom}(\bar{x}') \cap \operatorname{dom}(\bar{x}) = \emptyset$.

2. Let $(s', \bar{x}') = \tau(s, \bar{x})$ and $(t', \bar{y}') = \rho(t, \bar{y})$ such that $\operatorname{dom}(\bar{x}') \cap \operatorname{dom}(\bar{y}') = \operatorname{dom}(\bar{x}) \cap \operatorname{dom}(\bar{y}) = \emptyset$. Then $\tau \cup \rho$ is a well-defined bijection $\operatorname{dom}(\bar{x}) \cup \operatorname{dom}(\bar{y}) \to \operatorname{dom}(\bar{x}') \cup \operatorname{dom}(\bar{y}')$ and by definition of $(F \bullet G)$ $\tau \cup \rho$ relates $((s,t), \bar{x} \cup \bar{y})$ and $((s',t'), \bar{x}' \cup \bar{y}')$.

ζ is measurable: Recap from lemma 2 that any measurable set of $\mathcal{E}_X(F \bullet G)$ is of form $p[A]$ where $A \subseteq \widetilde{\mathcal{E}}_X F$ measurable. And:

$$\begin{aligned}
\zeta^{-1}(p[B]) &= \{([(s,\bar{x})], [(t,\bar{y})]) \mid \operatorname{dom}(\bar{x}) \cap \operatorname{dom}(\bar{y}) = \emptyset,\ [((s,t), \bar{x} \cup \bar{y})] \in p[A]\} \\
&= \{([(s,\bar{x})], [(t,\bar{y})]) \mid \operatorname{dom}(\bar{x}) \cap \operatorname{dom}(\bar{y}) = \emptyset,\ ((s,t), \bar{x} \cup \bar{y}) \in \operatorname{Orb}(A)\} \\
&= (p \times p)[E] \\
\text{where } E &= \bigcup_{\substack{V,W \\ V \cap W = \emptyset}} \Big\{((s, \bar{z}|_V), (t, \bar{z}|_W)) \mid ((s,t), \bar{z}) \in \operatorname{Orb}(A) \cap \widetilde{\mathcal{E}}_{X, V \cup W}(F \bullet G)\Big\}
\end{aligned}$$

The set E is a mere reordering of generalized tuple components from a measurable set and hence measurable. It is further known that images under p (and then also $p \times p$) are measurable.

It is easy to see that ξ and ζ are inverse to each other. □

D.1 Modeling algebraic data types

Now that the essential translations for species are set up, it is straightforward to give a model for a ADT in $\mathcal{M}$ compliant to the translation from section A.2.2.

Recap the general form of a Haskell ADT as in (A.1) with a single type argument a:

$$\texttt{data}\ T\ a = K_1\ t_{1,1}\ \ldots\ t_{1,k_1} \mid \ldots \mid K_n\ t_{n,1}\ \ldots\ t_{n,k_n}$$

Such a ADT corresponds to a species F of form

$$F \cong F_1 + \ldots + F_n$$

in the category **Spec**, where the F_i are species of form

$$F_i \cong F_{i,1} \bullet \ldots \bullet F_{i,k_i}.$$

Note that a $F_{i,j}$ can well be F or "contain" F in a sub-expression if T is recursive.[64]

Via lemma D.8, all the isomorphisms lift from **Spec** to $\mathcal{M}$ and via lemma D.10, "$+$" in **Spec** corresponds to "$\dot\cup$" in $\mathcal{M}$ and "$\bullet$" corresponds to "$\times$". Hence:

$$\mathcal{E}_X F \cong \dot{\bigcup_{i=1}^{n}} \bigtimes_{j=1}^{k_i} \mathcal{E}_X F_{i,j}$$

for any $X \in \mathcal{M}$. Defining the required functions is now straightforward:

Let $\iota_i : F_i \to F$ be the embeddings. Let X be some measurable space, e.g. a sort that already has an interpretation in $\mathcal{M}$ in the model $\mathcal{A}$. Define:

$$F^{\mathcal{A}}\ X := \mathcal{E}_X F$$
$$K_{i,X}^{\mathcal{A}} := \mathcal{E}_X \iota_i$$

Lemma D.10.2 ensures that ${K_{i,X}}^{\mathcal{A}}$ has the correct type (up to isomorphism).

For the `case` functions, let $f_i : \mathcal{E}_X F_{i,1} \times \ldots \times \mathcal{E}_X F_{i,k_i} \times Y \to Z$ be measurable functions, e.g. interpretations of terms, where Y and Z are measurable spaces. Y models the closure context as of section A.1.3.

Via lemma D.10.2, f_i can be viewed as a function $f_i : \mathcal{E}_X F_i \times Y \to Z$. By the universal property of the coproduct in $\mathcal{M}$, one receives

$$\dot{\bigcup_i} f_i \ : \ \dot{\bigcup_i} (\mathcal{E}_{F_i} X \times Y) \to Z.$$

The space on the LHS is isomorphic to $(\dot\bigcup_i \mathcal{E}_{F_i} X) \times Y$ which is again via lemma D.10.1 isomorphic to $\mathcal{E}_X F \times Y$. Lifting $\dot\bigcup_i f_i$ over these isomorphisms, one receives the desired function

$${\texttt{case}}^{\mathcal{A}}{}_{T,X,\bar f} : \mathcal{E}_F X \times Y \to Z.$$

It remains to check the axioms (A.2) and (A.3). The first states that any element of $\mathcal{E}_X F$ is given by one of the "constructors" $\mathcal{E}_X \iota_i$, which is clear by construction. The second states that `case` is "correct" in that a function can get back values from a constructor, which is clear by construction as well.

D.2 More on species and $\mathcal{M}$

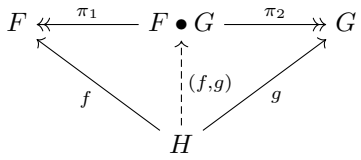
The mapping taking $f : F \to \mathcal{S}G$ to $\mathcal{E}_X f : \mathcal{E}_X F \to \mathcal{E}_X G$ can be made a functor as follows:

I already defined the function $\texttt{return} : F \to \mathcal{S}F$. One can also define $\texttt{join} : \mathcal{S}(\mathcal{S}F) \to \mathcal{S}F$ by

$$\texttt{join}_F : \quad \mathcal{S}(\mathcal{S}F) \to \mathcal{S}F$$
$$\texttt{join}_{F,U}(s \in F[W \subseteq V \subseteq U]) := s \in F[W \subseteq U] \subseteq \mathcal{S}F[U].$$

[64] This is the point where species do the "heavy lifting" of resolving the definitions of recursive ADTs.

Figure 13 Universal property of a potential product in **Kleisli**($\mathcal{S}$)



`join` "collapses the two instances of a layer" of $\mathcal{S}(\mathcal{S}F)$. Note that `join` is surjective, but not injective: For the same W and U, there are several possible choices of V unless $W = U$.

Having noted above that $\mathcal{S}$ is actually a functor **Spec** $\to$ **Spec**, it is easy to see that `return` and `join` are natural transformations $\mathtt{id} \to \mathcal{S}$ and $(\mathcal{S} \circ \mathcal{S}) \to \mathcal{S}$, respectively, i.e. that they commute with lifts of species morphisms to $\mathcal{S}$. Setting $\eta = \mathtt{return}$ and $\mu = \mathtt{join}$, it is easy to see that $\mathcal{S}$ is a monad as of [7, chap.r VI]. Alternatively, setting $\mathtt{fmap}\ f = \mathcal{S}f$ whenever $f : F \to G$ is a species morphism, one receives that the (equivalent) monad laws (*Mo1)–(*Mo5) from section 2 hold. So $\mathcal{S}$ is a monad.

If $f : F \to \mathcal{S}G$ and $g : G \to \mathcal{S}H$ are species morphisms, one receives a species morphism $(g \lll f) : F \to \mathcal{S}H$ as $\mathtt{join} \circ \mathtt{fmap}\ g \circ f$. "$\lll$" is called *Kleisli composition*.

The category that has species as objects, where an arrow $F \to G$ is a species morphism $F \to \mathcal{S}G$, where the identity morphism is `return` and where composition is done using "$\lll$" is called the *Kleisli category* **Kleisli**($\mathcal{S}$) of the monad $\mathcal{S}$.

It is easy to see that $\mathcal{E}_X(f \lll g) = \mathcal{E}_X f \circ \mathcal{E}_X g$ for f, g like above and that $\mathcal{E}_X \mathtt{return} = \mathtt{id}$. Hence, $\mathcal{E}_X$ is a functor **Kleisli**($\mathcal{S}$) $\to \mathcal{M}$ for any X.

Remark D.11. Note that **Kleisli**($\mathcal{S}$) still does not have $F \bullet G$ as the product: While the projections exist, it is not in general possible to find a (Kleisli) morphism (f, g) to make diagram 13 commute.

To see this, let $F = G = H = \mathbf{X}$ and $f = g = \mathtt{id}_{\mathbf{Kleisli}(\mathcal{S}),\mathbf{X}} = \mathtt{return}_{\mathbf{X}}$. For $|U| = 1$ then the map

$$\mathtt{return}_{\mathbf{X},U} : \mathbf{X}[U] \to \mathcal{S}\mathbf{X}[U]$$

would have to factor through $\mathcal{S}\mathbf{X}^2[U] = \emptyset$, but $\mathbf{X}[U] \neq \emptyset$.

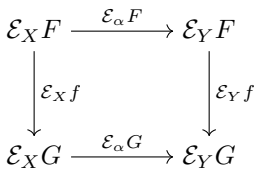
D.2.1 Lifting measurable functions

If species morphisms can be lifted to work on the "structure" part of $\mathcal{E}_X F$, it is natural to assume that measurable functions can work on the "content" X. This intuition turns out to be correct:

Lemma D.12. *Let $\alpha : X \to Y$ be a measurable function and let F be a species. Then α induces a measurable function*

$$\begin{aligned} \mathcal{E}_\alpha F : &\quad \mathcal{E}_X F \to \mathcal{E}_Y F \\ \mathcal{E}_\alpha F([(s, \bar{x})]) &:= [(s, \alpha \circ \bar{x})]\,. \end{aligned}$$

Figure 14 Universal property for $\mathcal{E}_\alpha : \mathcal{E}_X \to \mathcal{E}_Y$



The assignment is natural in F, i.e. one receives a natural transformation $\mathcal{E}_\alpha : \mathcal{E}_X \to \mathcal{E}_Y$.

Proof. Well-definedness and measurability: Consider the map

$$\widetilde{\mathcal{E}}_\alpha F : \quad \widetilde{\mathcal{E}}_X F \to \widetilde{\mathcal{E}}_Y F$$
$$\widetilde{\mathcal{E}}_\alpha F(s, \bar{x}) := (s, \alpha \circ \bar{x}).$$

This map is measurable: The maps $(\bar{x} \to \alpha \circ \bar{x})$ are just point-wise application of the measurable function α to the generalized tuple $\bar{x}$. When in doubt, use the universal property of the product. The map is also clearly compatible with "$\sim$". Hence, one receives $\mathcal{E}_\alpha F$ as its lift (lemma D.3.4).

Naturality: One has to show that diagram 14 commutes for any two species F and G and any natural transformation $f : F \to \mathcal{S}G$. But that is clear because $\mathcal{E}_X f$ and $\mathcal{E}_Y f$ operate only on the first component of a $(s, \bar{x})$ pair and $\mathcal{E}_\alpha F$ and $\mathcal{E}_\alpha G$ operate only on the second. □

To go one step further, note that the map $(\alpha \to \mathcal{E}_\alpha)$ is itself functorial: We have that $\mathcal{E}_{\mathtt{id}_X} = \mathtt{id}_{\mathcal{E}_X}$ for any X and $\mathcal{E}_{(\beta \circ \alpha)} = \mathcal{E}_\beta \circ \mathcal{E}_\alpha$ for $\alpha : X \to Y$ and $\beta : Y \to Z$.

Hence, $\mathcal{E}$ is a functor from the category of measurable spaces and maps to the category of functors $\mathbf{Kleisli}(\mathcal{S}) \to \mathcal{M}$ and their natural transformations. Short: $\mathcal{E} : \mathcal{M} \to \mathcal{M}^{\mathbf{Kleisli}(\mathcal{S})}$.

By abstract arguments, one can also view $\mathcal{E}$ as a functor $\mathbf{Kleisli}(\mathcal{S}) \to \mathcal{M}^{\mathcal{M}}$: $\mathcal{E}$ assigns to any species F the functor $\mathcal{E}F$ maps $X \in \mathcal{M}$ to $\mathcal{E}_X F \in \mathcal{M}$ and mapping $\mathbf{Kleisli}(\mathcal{S})$-morphisms (in particular species morphisms) to natural transformations of these functors.

So the question whether "every ADT is a functor" can not only be answered affirmative, but even every *species* defines a functor *and* the assignment is itself functorial on $\mathbf{Kleisli}(\mathcal{S})$.

Remark D.13. As a final remark, note that none of the proofs here used complementation in the σ-algebra. Hence, the category $\mathcal{M}$ of measurable spaces could be replaced by the category **Top** to receive topological spaces and continuous functions instead.

References

[1] J. C. Hull, *Options, Futures and Other Derivatives*, 8th ed. Boston, MA, USA: Pearson/Prentice Hall, 2012.

[2] H. Föllmer and A. Schied, *Stochastic Finance: An Introduction in Discrete Time*, 2nd ed., ser. de Gruyter Studies in Mathematics. Berlin, Germany: Walter de Gruyter, 2004.

[3] S. Peyton Jones and J.-M. Eber, "How to write a financial contract," in *The Fun of Programming*, ser. Cornerstones of Computing, J. Gibbons and O. de Moor, Eds. Palgrave Macmillan, 6 2005. [Online]. Available: http://research.microsoft.com/en-us/um/people/simonpj/papers/papers.html

[4] 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. [Online]. Available: http://research.microsoft.com/en-us/um/people/simonpj/papers/papers.html

[5] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries – The Revised Report*. Cambridge, England: Cambridge University Press, 2003. [Online]. Available: http://www.haskell.org/haskellwiki/Language_and_library_specification

[6] P. Wadler, "Monads for functional programming," in *Marktoberdorf Summer School on Program Design Calculi*, ser. NATO ASI Series F: Computer and systems sciences, vol. 118. Springer, 1992, also in J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995. [Online]. Available: http://homepages.inf.ed.ac.uk/wadler/topics/monads.html

[7] S. Mac Lane, *Categories for the Working Mathematician*, 2nd ed., ser. Graduate Texts in Mathematics. New York, NY, USA: Springer, 1998.

[8] P. Blackburn, M. de Rijke, and Y. Venema, *Modal Logic*. New York, NY, USA: Cambridge University Press, 2001.

[9] T. Jech, *Set Theory*, 3rd ed., ser. Springer Monographs in Mathematics. Berlin, Germany: Springer, 2002.

[10] R. J. Aumann, "Borel structures for function spaces," *Illinois Journal of Mathematics*, vol. 5, no. 4, pp. 614–630, 12 1961. [Online]. Available: http://projecteuclid.org/euclid.ijm/1255631584

[11] G. Nedoma, "Note on generalized random variables," in *Transactions of the First Prague Conference on Information Theory, Statistical Decision Functions, Random Processes 1956*, J. Kozesnik, Ed. Czechoslovak Academy of Sciences, 1957.

[12] E. Schechter, *Handbook of Analysis and Its Foundations*. San Diego, CA, USA: Academic Press, 1997.

[13] A. S. Kechris, *Classical Descriptive Set Theory*, ser. Graduate Texts in Mathematics. New York, NY, USA: Springer, 1995.

[14] S. Peyton Jones, *The implementation of functional programming languages*. Hertfortshire, UK: Prentice Hall, 1987. [Online]. Available: http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/

[15] M. Manzano, *Extensions of First-Order Logic*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge, UK: Cambridge University Press, 1996.

[16] The GHC Team, "The glorious glasgow haskell compilation system user's guide, version 7.4.1," Jun. 2014. [Online]. Available: http://www.haskell.org/ghc/docs/7.4.1/html/users_guide/

[17] A. Yakeley, "time: A time library," http://hackage.haskell.org/package/time-1.4.2, March 2014.

[18] B. A. Yorgey, "Species and functors and types, oh my!" in *Proceedings of the third ACM Haskell symposium on Haskell*, ser. Haskell '10. New York, NY, USA: ACM, 2010, pp. 147–158. [Online]. Available: http://www.cis.upenn.edu/~byorgey/pub/species-pearl.pdf