Declarative Pearl: Rigged Contracts

Alexander Vandenbroucke[1] [0000−0002−5683−1570] and Tom Schrijvers[2] [0000−0001−8771−5559]

[1] Unaffiliated
alexander.vandenbroucke@sc.com
[2] KU Leuven, Leuven, Belgium
tom.schrijvers@kuleuven.be

Abstract. Over 20 years ago, Peyton Jones et al. embarked on an adventure in financial engineering with their functional pearl on "Composing Contracts". They introduced a combinator library—a domain-specific language—for precisely describing complex financial contracts and a formal denotational semantics for computing their value, for which they briefly sketched an implementation.
This paper reworks the design of their library to make the central datatype of contracts less ad-hoc by giving it a well-understood algebraic structure: the semiring. Then, interpreting a contract's worth as a generic semiring homomorphism directly gives rise to a natural semantics for contracts, of which computing the (monetary) value is but one instance.

Keywords: semiring · financial contract · domain-specific language.

1 Introduction

Consider the following contract from "Composing Contracts" [6]:

```
C The right to choose on 30 June 2000 between:
  C1 Both of:  C11 Receive $100 on 29 Jan 2001.
               C12 Pay $105 on 1Feb 2002.
  C2 An option exercisable on 15 Dec 2000 to choose one of:
    C21 Both of:  C211 Receive $100 on 29 Jan 2001.
                  C212 Pay $106 on 1 Feb 2002.
    C22 Both of:  C221 Receive $100 on 29 Jan 2001.
                  C222 Pay $112 on 1 Feb 2003.
```

This simplified contract is representative of those that commonly occur in the finance industry. A key insights is that larger contracts, such as `C`, are created by composing smaller contracts, such as `C1` and `C2`.

The finance industry employs an extensive vocabulary for describing specific forms of financial contracts (swaps, futures, caps, floors, American options, and European options, to list but a few). As Peyton Jones et al. [6] say "Treating each of these individually is like having a large catalogue of prefabricated components. The trouble is that someone will soon want a contract that is not in the catalogue." The benefit of realising and exploiting the compositional nature of contracts is that we can describe and reason about a vast class of complex contracts with only a small set of primitive combinators. Indeed, with only ten combinators, Peyton Jones et al. manage to express a wide variety of contracts.

The design of a combinator library can be helped immensely by relying on algebraic abstractions (e.g., [8]). Therefore, a natural question is if there is a suitable abstraction for financial contracts. This paper focuses on such an abstraction: the semiring (also called a *rig*). A semiring is a set equipped with two operations and two identities for those operations, satisfying certain axioms. Semirings capture a large variety of concepts, e.g., natural, integer and real numbers, polynomials, gradients, probabilities, and tropical semirings.

The key idea of this paper is that contracts also form a semiring. The two primary ways of combining contracts ("Choose between" and "Both Of") are associative and commutative. Both operations have identities, the first of which is the second's annihilator. Finally, "Both Of" distributes over "Choose Between".

To give a precise meaning to the monetary value of a contract, Peyton Jones et al. [6] define a denotational semantics of the combinators in terms of so-called "value processes", i.e., time-varying probabilistic processes. Here too, we can take inspiration from mathematics, by reimagining the semantics as a *universal semiring homomorphism*, i.e. a structure-preserving function from contracts to any semiring (equipped with inverses). In other words, the contract semiring is the initial object with respect to such semirings. In essence, this defines a family of correct-by-construction interpretations for contracts where the particular interpretation depends on the targeted semiring. Thus, we can instantiate new interpretations for contracts, simply by plugging in semirings from the literature. Notably, the tropical semiring coincides with the original semantics.

We make the following changes to Peyton Jones et al. [6]'s contract library: (1) a new annihilator contract, `expired`; (2) a new combinator `both`, that replaces `and`; and (3) a small change to the meaning of *expiry date* and the related `truncate`-primitive. These changes make contracts into a semiring, without losing expressivity. Reviewer 1 characterizes them as follows: "*Computationally, they are every bit as reasonable as the originals, syntactically they are no harder, and semantically they are much more understandable and satisfying.*"

Moreover, we give a universal definition of a homomorphism from contracts to any semiring that has a multiplicative group, i.e., whose multiplicative operation is invertible. Assuming that such a semiring admits a useful financial model, this definition *subsumes* the original denotational semantics. This paper is written in Literate Haskell, but a more complete implementation of the combinator library is available at https://github.com/tschrijv/RiggedContracts.

2 The Contract Library

This section presents the contract combinator library, which is inspired by—but not identical to—that of Peyton Jones et al. Our running example contract is the *zero-coupon bond*, one of the simplest contracts in the finance industry:

```
zcb :: Time -> Double -> Currency -> Contract obs
c0 = zcb (date "21 Apr 2020") 100 EUR
```

Contract `c0` entitles its holder to receive 100 EUR on the April 21, 2020. The `zcb` function takes a time, a `Double` amount and a `Currency`; and it returns a value of type `(Contract obs)`.[3] A `Currency` is a sum type of different currencies, e.g.:

```
data Currency = EUR | GBP | ...
```

We assume a given function `date :: String -> Time` that turns a textual representation of a date into a value of the type `Time`, which is defined in Section 3.1.

2.1 Acquisition Date and Expiry Date

To give a precise description of the contract combinators, we must first define two technical notions: the *acquisition time*[4] and the *expiry date*. For the purposes of this paper, a contract is a legally binding agreement between the holder of the contract and another party. *Acquiring* a contract means that the holder enters into a legally binding agreement with the other party, with legal consequences for both parties that stem from the rights and obligations it mentions. These consequences depend on the *acquisition time*, the time at which the contract is acquired. For example, the contract `c0` above is worth a lot less if it is acquired on the 22$^{\text{nd}}$ of April, 2020 than if it is acquired on the 19$^{\text{th}}$ of the same month, because obligations and rights before the acquisition time have no effect.

Complementary to the acquisition time is the contract's *expiry date*. The expiry date is defined as *the earliest point in time at which a contract can no longer be acquired*. This differs subtly from the concept of a *horizon* as defined by Peyton Jones et al. [6], which is *the latest time at which it can be acquired*. We use the term "expiry date" rather than "horizon" to emphasise this distinction. This small—but crucial—difference is necessary to equip contracts with a semiring (see Sections 3.2 and 4.3). A contract's expiry date is an innate property that is completely specified by its definition (see Figure 1). However, a contract's consequences may extend well beyond its expiry date. Consider a contract that confers "the right to decide on Dec 26, 2020 whether or not to acquire contract C". This contract must be acquired before Dec 26, 2020—its expiry date—but the underlying contract C may have consequences much later than Dec 26, 2020. This kind of contract is called an *option*.

Figure 1 describes each primitive contract combinator in an informal manner.[5] The relevant parts of the description that differ from Peyton Jones et al.'s due to our definition of the expiry date are underlined.

2.2 Discount Bonds

To illustrate the combinator library, let us reconsider the zero-coupon bond, `zcb`, which, it turns out, is not a primitive. It is defined as follows:

[3] The meaning of the `obs` parameter is explained in Section 2.2.
[4] Also called the acquisition *date*.
[5] Backquotes turn a function into an infix operator, e.g., `` x `f` y = f x y ``.

`expired :: Contract obs`
: This contract expires at the *epoch*, the first moment in time. Because the contract is always expired, it is not acquirable.

`zero :: Contract obs`
: `zero` is a contract that may be acquired at any time. It conveys neither rights nor obligations, and never expires.

`one :: Currency -> Contract obs`
: `(one k)` is a contract that immediately pays the holder one unit of the currency `k`. The contract never expires.

`give :: Contract obs -> Contract obs`
: To acquire `(give c)` is to acquire all `c`'s rights as obligations and vice versa. It expires when the underlying contract expires.

`both :: Contract obs -> Contract obs -> Contract obs`
: If you acquire `both c1 c2`, then you immediately acquire both `c1` and `c2` unless either `c1` or `c2` has expired, in which case you acquire neither. The composite contract expires when either `c1` or `c2` expires.

`or :: Contract obs -> Contract obs -> Contract obs`
: If you acquire `(c1 ‘or‘ c2)` then you must immediately acquire `c1` or `c2` (*but not both*). If either has expired, that one cannot be acquired. When both have expired, the compound contract has expired.

`truncate :: Time -> Contract obs -> Contract obs`
: `(truncate t c)` is exactly like `c` except that its expiry date is the earlier of `t` and the expiry date of `c`. Notice that `truncate` limits only the *possible acquisition dates* of `c`; it does not truncate `c`'s rights or obligations, which may extend beyond `t`.

`thereafter :: Contract obs -> Contract obs`
: If you acquire `(c1 ‘thereafter‘ c2)` and `c1` has not expired, then you must acquire `c1`. If `c1` has expired and `c2` has not, you must acquire `c2`. The compound contract expires when `c1` and `c2` expire. (Called “`then`” in the original paper, a reserved Haskell keyword.)

`scale :: obs -> Contract obs -> Contract obs`
: If you acquire `(scale o c)`, then you immediately acquire a contract just like `c`, except that all rights and obligations of `c` are multiplied by the value of the observable `o` at the moment of acquisition. The scaled contract expires when the underlying contract expires.

`get :: Contract obs -> Contract obs`
: If you acquire `(get c)` then you must acquire `c` just before it expires. The new contract expires when the underlying contract expires.

`anytime :: Contract obs -> Contract obs`
: If you acquire `(anytime c)` then you must acquire `c`, but you can do so at any time (from the acquisition of `(anytime c)` onwards) before `c` expires. The new contract expires when the underlying contract expires.

Fig. 1: Contract Combinators

```
zcb :: Time -> r -> Currency -> Contract (Time -> r)
zcb time amount currency = scaleK amount (get (truncate time (one currency)))
```

At its core is the `one` contract:

```
one :: Currency -> Contract obs
```

Acquiring `(one EUR)` at any time immediately gives you €1. However, suppose instead that we want to receive €100 at a specific time, and not earlier. First, to fix the time, we combine `get` and `truncate`, to get a contract that gives you €1 at a specific time `t`:

```
get (truncate t (one EUR))
```

The `truncate` combinator trims `(one EUR)`'s expiry date to `t`, so that it can only be acquired before `t`, and `get` forces `truncate t (one EUR)` to be acquired at the last possible moment, i.e. *just before* `t`. The combined effect is the desired one, namely that by acquiring this contract at any time before `t`, the holder will receive one euro at `t`.

Second, to receive €100, not €1, we must scale up the contract by a factor of 100. This scaling is achieved with the auxiliary combinator `scaleK`, which builds on the primitive combinator `scale`:

```
scaleK :: r -> Contract (Time -> r) -> Contract (Time -> r)
scaleK x c = scale (const x) c
```

The contract `(scale obs c)`, when acquired at time `t`, scales all the rights and obligations of `c` by the value of the *observable* `obs` at time `t`. Observables are time-varying quantities like a particular stock price, interest rate, or even temperature. For simplicity, this paper represents observables as functions of type `(Time -> r)`. The `scaleK` combinator simply uses a constant function, i.e. scales with a constant factor.

2.3 Composing Contracts

So far, we have seen combinators that create new contracts by modifying a single contract in some way (e.g., truncating its expiry date or scaling its value). Another way to create contracts is by composing several into a larger contract. A straightforward way of combining two contracts `c1` and `c2` is by creating a new contract that confers the rights and obligations of both `c1` and `c2`. This is accomplished by the `both` combinator (see Figure 1):

```
both :: Contract obs -> Contract obs -> Contract obs
```

Upon acquiring `both c1 c2`, you must immediately acquire both `c1` and `c2`, unless either `c1` or `c2` has expired, in which case you acquire *neither*. The compound contract expires when either `c1` or `c2` expires. For example, the following is a contract that entitles you to receive €100 at time `t1`, and €100 more at `t2`:

```
both (zcb t1 100 EUR) (zcb t2 100 EUR)
```

Another way to combine contracts is to introduce a choice between two contracts: upon acquiring the contract `(c1 ‘or‘ c2)` you must immediately acquire either `c1` (if it has not expired) or `c2` (if it has not expired), but not both. The compound contract expires when both `c1` and `c2` have expired. This operator is commutative: choosing between `c1` and `c2` is equivalent to choosing between `c2` and `c1`.

For example, the following contract entitles you to receive €100 at time `t1`, or €200 at time `t2`:

```
zcb t1 100 EUR ‘or‘ zcb t2 200 EUR
```

The available choices are determined by the acquisition time: if you acquire the contract above after `t1`, but before `t2`, only the second contract is still available. You cannot choose to do nothing.

Nevertheless, the option to do nothing is useful to have, and it is captured by the `zero` contract, which confers neither rights nor obligations. For example, it allows a style of contract, called a *European option* that bestows the right to decide, at a particular time, whether or not to acquire some underlying contract:

```
european :: Time -> Contract obs -> Contract obs
european t c = get (truncate t (c ‘or‘ zero))
```

We see that we first introduce a choice between acquiring `c` and doing nothing (`zero`). Next, we `truncate` the expiry date of the choice to `t` and use `get` to enforce that the choice is made just before `t`.

Recall that contracts are agreements between *two* parties. Thus, it seems rather unfair that the `one` combinator only pays the holder, and not the other party. This situation is reversed by the `give` combinator: `(give c)` is the contract `c`, with the rights and obligations reversed. For example, the following is a contract whose holder receives €100 at time `t1`, and *pays* €200 at time `t2`:

```
both (zcb t1 100 EUR) (give (zcb t2 200 EUR))
```

Note that `give` also changes who makes the choices. E.g., in the following,

```
give (european t1 (give (zcb t2 200 EUR)))
```

the other party decides whether the holder of the contract receives €200.

Both vs. And Instead of `both`, we could define a similar combinator `and`: if you acquire `(c1 ‘and‘ c2)`, you acquire both `c1` (unless it has expired) and `c2` (unless it has expired), and the compound contract expires only when both `c1` and `c2` have expired. That is, one contract expiring does not prevent you from acquiring the other contract. The `both` combinator enforces a stronger tie between two contracts: For instance, suppose contract `c1` obliges the holder to pay a certain amount, and `c2` obliges the holder to receive a certain amount, then `(both c1 c2)` ensures that no money is received (`c2`) without the required payment (`c1`). On the other hand, `(c1 ‘and‘ c2)` may seem more flexible.

However, unlike in Peyton Jones et al.'s library, where `and` is defined as a primitive, in our library, the behaviour of `and` can instead be recovered from `both` and `or` by acquiring `(both c1 c2)` before it expires, or afterwards picking up the remaining contract with `(c1 ‘or‘ c2)`:

```
and :: Contract obs -> Contract obs -> Contract obs
c1 ‘and‘ c2 = (both c1 c2) ‘thereafter‘ (c1 ‘or‘ c2)
```

The `thereafter` combinator (see Figure 1) composes contracts sequentially: it blocks the acquisition of a contract until another contract has expired. More precisely: if you acquire `(c1 ‘thereafter‘ c2)`, you must acquire `c1`, *unless* `c1` has expired, in which case you must acquire `c2`. Consider, for instance:

```
zcb t1 200 EUR ‘thereafter‘ zcb t4 300 EUR
```

If it is acquired before `t1`, it entitles you to receive €200 on `t1`. If it is acquired on or after `t1`, but before `t4`, it entitles you to receive €300 on `t4`.

Choosing When The `or` combinator allows the holder to choose which of two contracts to acquire. Conversely, the `anytime` combinator allows the holder to choose *when* to acquire a contract. More precisely, if you acquire `(anytime c)`, then you *must* acquire `c`, but you are free to do so at any time after the acquisition date and before `c` expires. This allows the contract language to express an *American option.* Unlike the European option, an American option not only allows you to decide whether to acquire an underlying contract, but also *when* to do so (within a specific time interval):

```
american :: Time -> Time -> Contract obs -> Contract obs
american t1 t2 c = beforeT1 ‘thereafter‘ afterT1 where
  beforeT1 = get (truncate t1 afterT1)
  afterT1  = anytime (truncate t2 (zero ‘or‘ c))
```

There are two parts to this contract. The first part, `beforeT1`, prevents the acquisition of `c` *until* `t1`: if the option is acquired before `t1`, all it does is ensure you acquire `afterT1` at `t1`. Otherwise, if the option is acquired after `t1`, `afterT1` is acquired directly. This contract then allows you to choose whether and when to acquire `c`—until `t2`, when the option expires.

3 Instant Semiring, Just Add Expired

Having introduced all the necessary combinators, we now recall the definition of a semiring and then explain how to equip contracts with a semiring structure.

3.1 Definition of a Semiring

A semiring is an algebraic structure defined as follows:

Definition 1 (Semiring). *A semiring* $(R, +, \times, 0, 1)$ *is a set* R *equipped with two operations* $+$ *and* $\times$ *and elements* $0, 1 \in R$ *such that* $+$ *and* $\times$ *are associative, with identities* 0 *and* 1*, respectively. Additionally,* $+$ *is commutative,* 0 *is the annihilator for* $\times$*, and* $\times$ *distributes over* $+$ *on the left and right.*

That is, $+$ and $\times$ are *monoids* (they are associative and each have a neutral element). Additionally, $+$ *commutes*, but $\times$ does not have to. For instance, square-matrix multiplication is a semiring, even though matrix multiplication is not commutative (unless the matrices are invertible). Annihilation and distributivity ensure that $+$ and $\times$ are compatible.

In Haskell, we capture semirings in the following `Semiring` type class:

```
class Semiring r where
  nil   :: r
  unit  :: r
  plus  :: r -> r -> r
  times :: r -> r -> r
```

```
instance Semiring Double where
  nil   = 0
  unit  = 1
  plus  = (+)
  times = (*)
```

Numeric semirings The natural numbers, integers and real numbers form semirings, with their standard notions of addition and multiplication. For instance, we can define the above instance for `Double`.[6]

Tropical Semirings A more exotic variant of a semiring is the Tropical Semiring. The *max(resp. min)-tropical semiring* consists of the real number line extended with negative (resp. positive) infinity. Addition is defined by taking the maximum of its arguments, and multiplication is addition of the extended real numbers. In Haskell, we provide a slightly more general definition:

```
data Max a = NegInfty | Max a    deriving (Eq,Ord,Show,Functor)
data Min a = Min a    | PosInfty deriving (Eq,Ord,Show,Functor)

instance (Ord a, Semiring a) => Semiring (Max a) where
  nil  = NegInfty
  unit = Max nil
  plus = max
  Max a `times` Max b = Max (a `plus` b)
  _ `times` _         = NegInfty
-- instance Semiring Min omitted for brevity
```

Time Semiring The natural numbers extended with positive infinity possess an alternative semiring, by setting $+$ to max and $\times$ to min. This proves to be a convenient definition of time:

```
data Time = Finite Int | Infinite deriving (Eq,Ord,Show)

instance Semiring Time where
  nil   = epoch           ;  unit  = Infinite
  plus  = max             ;  times = min
```

Thus, `Time` is an ordered series of discrete points, beginning at the `epoch`, and extending infinitely into the future.

[6] In reality, floating point numbers such as `Double` do violate semiring axioms due to rounding errors. Here, we stick with `Doubles` for simplicity.

```
epoch :: Time
epoch = Finite 0

previous :: Time -> Maybe Time
previous (Finite t) | t > 0 = Just (Finite (t - 1))
previous _          = Nothing
```

Every point in time has a `previous` time, except `epoch` and `Infinite`. The major implication is that a contract that expires at the epoch is unobtainable, since a contract must be acquired *strictly before* its expiry date.

3.2 The Contract Semiring

The contract combinators `or` and `both` form a semiring. The former is $+$, the latter $\times$. From their informal descriptions, it is quite easy to see that both operators are associative and commutative, and that `both` distributes over `or`. These properties can be proved formally using the semantics defined in Section 4.

The identity contract for `both` is `zero`. Intuitively, acquiring `(both c zero)` acquires exactly the rights and obligations of `c`, with exactly the same expiry date, since `zero` has neither rights nor obligations, and has an infinite expiry date. Symbolically,

$$\texttt{both c zero} = \texttt{c}.$$

The identity for `or` is the contract `expired`, which expires at the epoch, meaning that actually obtaining `expired` is impossible. To see why this is the neutral element, consider that upon acquiring ``(c `or` expired)``, one must acquire either `c` or `expired`. Because `expired` expires at the `epoch`, the only permissible option is to acquire `c`, symbolically,

$$\texttt{c `or` expired} = \texttt{c}.$$

Additionally, `expired` annihilates any other contract with respect to `both`: because `(both expired c)` expires when the most short-lived contract expires, it expires when `expired` does, at the epoch. But this exactly matches the definition of `expired`, symbolically,

$$\texttt{both c expired} = \texttt{expired}.$$

These four combinators give rise to the following semiring instance:

```
instance Semiring (Contract obs) where
  nil   = expired     ;  unit  = zero
  plus  = or          ;  times = both
```

3.3 Beyond Semirings: Groups

A group is a fundamental algebraic concept which captures the notion that a binary operator is invertible. In particular, it is useful to formalise the idea that some contracts cancel each other, for example `(one EUR)` and `(give (one EUR))`. Formally, a group is defined as:

Definition 2 (Group). *A group $(R, +, 0, -)$ is a monoid $(R, +, 0)$ equipped with an operation $(-) : R \to R$ such that for all $a \in R$: $a + -a = 0 = -a + a$.*

Groups pervade mathematics. For instance, integer or real addition form groups with negation, and real multiplication forms a group with the reciprocal, to give but two straightforward examples. Generally, two kinds of groups can be distinguished in a semiring, depending on whether they arise from the additive operator $(+)$ or from the multiplicative operator $(\times)$.

For example, the natural numbers have neither additive nor multiplicative inverses; the integers have an additive inverse, but no multiplicative inverse; and the rationals have both, as do the reals. Tropical semirings do not have an additive group, but they have a multiplicative one, which is the additive group of the underlying semiring. The `Time` semiring has no groups.

Formally, these groups are defined as follows:

Definition 3. *Let $(R, +, \times, 0, 1)$ be a semiring.*

- *The semiring has an* additive group *if there exists an additive inverse $(-)$ such that $(R, +, 0, -)$ is a group.*
- *The semiring has a* multiplicative group *if there exists a multiplicative inverse $(\cdot)^{-1}$ such that $(R\backslash\{0\}, \times, 1, (\cdot)^{-1})$ is a group.*
- *A semiring that has an additive group is called a* ring*. (For this reason a semiring is sometimes also called a* rig*, i.e. a ring without* negatives*.)*
- *A semiring with both an additive and a multiplicative group is called a* field*.*

Notice that an additive group has all of R as its underlying set, but a multiplicative group only has R without 0. This situation is analogous to the real numbers and other fields, where division by zero is undefined.

The `Contract` semiring has a (multiplicative) group: the inverse of a contract `c` is the contract `(give c)`, which reverses the rights and obligations of the two parties. Indeed, if one party acquires `c`, the other party acquires `(give c)` at the same time. Note that `give` changes the primary actor of a contract: if one party makes a choice (e.g. `c1 ‘or‘ c2`) in `c`, the other party is entitled to make that choice in `(give c)`. This means that acquiring `(both c (give c))` is equivalent to acquiring `zero`, since acquiring `(give c)` cancels out the rights and obligations of any contract `c` that is not `expired`.[7]

In Haskell, we define the type classes `Additive` and `Multiplicative` for semirings with an additive and multiplicative group, respectively:

```
class Semiring r => Additive r         where neg :: r -> r
class Semiring r => Multiplicative r   where inv :: r -> r
instance Multiplicative (Contract obs) where inv = give
```

Section 4.4 and below also need instances for `Double` and `Max` (`Max` has a multiplicative group if the underlying semiring has an additive group).[8]

[7] `give` need not cancel out `expired`, since the annihilator of the semiring is excluded from the multiplicative group.

[8] The partial `inv` is acceptable since the annihilator, `NegInfty`, need not be invertible.

```
instance Additive Double       where neg = negate
instance Multiplicative Double where inv = recip
instance (Ord r, Additive r) => Multiplicative (Max r) where
  inv (Max x) = Max (neg x)
```

All is now in place to formally define the denotational semantics of a contract.

4 Denotational Semantics: Expiry Date and Worth

This section presents the denotational semantics of the contract language as implemented in Haskell, using a *deep embedding* [4]. In this style of embedding, the abstract syntax tree of the contract language is explicitly reified as a Haskell data type `Contract` that has a constructor for each primitive:

```
data Contract obs
  = Zero
  | Both (Contract obs) (Contract obs)
  | Or (Contract obs) (Contract obs)
  | Give (Contract obs)
  | Truncate Time (Contract obs)
  | Thereafter (Contract obs) (Contract obs)
  | One Currency
  | Scale obs (Contract obs)
  | Get (Contract obs)
  | Anytime (Contract obs)
```

Observe that `expired` is not a primitive, it can be defined as:

```
expired = truncate epoch zero
```

The appendix derives this equation from the denotational semantics using straightforward equational reasoning. The denotational semantics itself consists of two distinct parts: a contract's *expiry date* and its *worth* (e.g., monetary value).

4.1 A Contract's Expiry Date

The `expiry` function in Figure 2 calculates the earliest point at which its argument contract can no longer be acquired. The definition has a case for each constructor of Figure 1. Moreover, the expiry date of `expired` is:

$$\begin{aligned}\texttt{expiry expired} &= \texttt{expiry (truncate epoch zero)}\\ &= \texttt{min (expiry zero) epoch}\\ &= \texttt{epoch} \qquad \text{(E1)}\end{aligned}$$

Recall that `Time` is a semiring, where `nil` is `epoch`, `unit` is `Infinity`, `times` is the minimum, and `plus` is the maximum. Then, looking at Equation (E1) and lines 2, 3 and 4 in Figure 2, it follows that `expiry` preserves the semiring structure: it (recursively) maps the `nil`, `unit`, `times` and `plus` of the contract semiring

```
expiry :: Contract obs -> Time
expiry Zero                = Infinite
expiry (Both c1 c2)        = min (expiry c1) (expiry c2)
expiry (Or c1 c2)          = max (expiry c1) (expiry c2)
expiry (Give c)            = expiry c
expiry (Truncate t c)      = min (expiry c) t
expiry (Thereafter c1 c2)  = max (expiry c1) (expiry c2)
expiry (One c)             = Infinite
expiry (Scale o c)         = expiry c
expiry (Get c)             = expiry c
expiry (Anytime c)         = expiry c
```

Fig. 2: Expiry Date

to their counterparts in the `Time` semiring. Such a structure-preserving function is called a (semiring) *homomorphism*. Because it preserves the compositional nature of its argument, a homomorphism enables modular equational reasoning about contracts.

4.2 The Bottom Line

As a rule, a contract is acquired because it is worth something to its holder. How much depends on the real-world financial context and the time at which it is acquired. This value is computed by the function `worth` (Figure 3). The main work is done by the local function `go` (lines 11–23). By design, `go` only returns non-`nil` values *before the contract's expiry date*, and `nil` ever after:

$$\texttt{go c t} = \texttt{nil} \Leftarrow \texttt{t} \geq \texttt{expiry c} \tag{W1}$$

Before the contract expiry date, `go` satisfies the following properties:

$$\texttt{go expired t} = \texttt{nil} \tag{W2}$$

$$\texttt{go zero t} = \texttt{unit} \tag{W3}$$

$$\texttt{go (both c1 c2) t} = \texttt{go c1 t `times` go c2 t} \tag{W4}$$

$$\texttt{go (c1 `or` c2) t} = \texttt{go c1 t `plus` go c2 t} \tag{W5}$$

$$\texttt{go (give c) t} = \texttt{inv (go c t)} \tag{W6}$$

Equations (W2–W6) state that `go` is a *multiplicative* semiring homomorphism. That is, the worth of a contract is always interpreted in a generic multiplicative semiring, such that `go` preserves the contract's semiring structure (W2–W5) and multiplicative inverses (W6). Equation (W2) is somewhat redundant: it follows from Equation (W1) that `go` maps `expired` to `nil`.

Implementation Let us now look at each line of `go` in detail. *Line 9* implements Equation (W1). *Lines 10–13* implement Equations (W3–W6). In *Line 14* the

```
1  data Financial r obs =
2    Financial { exch  :: Currency -> Time -> r
3              , disc  :: Time -> r -> Time -> r
4              , snell :: (Time -> r) -> Time -> r
5              , eval  :: obs -> Time -> r -> r     }
6
7  worth :: Multiplicative r => Financial r obs -> Contract obs -> Time -> r
8  worth (Financial exch disc snell eval) = go where
9    go c                  time | time >= expiry c = nil
10   go Zero               time = unit
11   go (Both c1 c2)       time = go c1 time ‘times‘ go c2 time
12   go (Or c1 c2)         time = go c1 time ‘plus‘ go c2 time
13   go (Give c)           time = inv (go c time)
14   go (Truncate t c)     time = go c time
15   go (Thereafter c1 c2) time | time < expiry c1 = go c1 time
16                              | otherwise        = go c2 time
17   go (One k)            time = exch k time
18   go (Scale o c)        time = eval o time (go c time)
19   go (Get c)            time | Just t <- horizon c = disc t (go c t) time
20                              | otherwise = nil
21   go (Anytime c)        time | Just t <- horizon c = snell (go c) time
22                              | otherwise = nil
23   horizon = previous . expiry
```

Fig. 3: Worth

worth of `truncate t c` is simply the worth of `c`, because line 9 has already checked that `time` is less than the expiry date. In *Lines 15 and 16* the worth of `(c1 ‘thereafter‘ c2)` is the worth of `c1` if `time` has not yet passed the expiry date of `c1`, otherwise it is the worth of `c2`.

So far, all the cases have been nicely generic. However, to define the remaining cases, we must introduce a financial model into our abstract mathematics. This model consists of the data type `Financial`, which contains all the finance-specific know-how. This knowledge is expressed in a specific currency `k'`, for instance `exch k t` is the value, expressed in `k'` of one unit of currency `k`, at time `t`. There are some properties that the components `exch`, `disc` and `snell` must satisfy. For brevity, we refer the interested reader to the original [6] for their definitions. The meaning of these components is best explained by their usage in lines 17–23. For instance, *line 17* says that the worth of obtaining one unit of a specific currency `k` is `exch k t`, the value at time `t` of one unit of `k` expressed in the financial model's currency `k'`. Next, *line 18* defines the worth of `(scale o c)` by evaluating the *observable* `o` at `time` and multiplying it with the `c`'s worth at `time`. This scaling operation is captured by the function `eval`.

The last two cases are identical to the version in the original paper: The penultimate case computes the worth of `(get c)` (*line 19*), which is the value of the contract `c` when it is acquired (`t`), but discounted to the current time

(`time`). The function `disc` models this style of interest evolution: given a time `t` and a value `v` at time `t'`, `(disc t' v t)` is the interest-rate discounted value of `v` at time `t` expressed in currency `k'`.

The final case computes the worth of `(anytime c)` (*line 21*). Upon acquiring `(anytime c)`, the primary party *must* acquire `c`, but they can do so at any time between the acquisition and the expiry date of `c`. The idea is that it allows one to choose to acquire `c` when it is most valuable, possibly based on exchange and interest rates, share prices, etc. Determining the optimal time to acquire `c` is complicated, and requires finding the *snell-envelope* of `c` (*line 21*). The implementation of `disc` and `snell` is beyond the scope of this paper. The original paper briefly sketches one possible implementation in Haskell.

4.3 Comparison with the Original

There are two main difference between the original library and ours. Firstly, as mentioned in Section 2.3, their `and` combinator is less expressive than our `both` combinator. Secondly, the original library cannot express the `expired` contract,[9] which has no horizon, and, as a consequence, lacks the corresponding semantic notion: the `nil` of the semiring. Lacking this notion, their worth function is partial; it is simply undefined where ours is `nil`.

4.4 Executable Semantics

The code presented in Figure 3 is a denotational semantics; its primary purpose is to give a precise, formal, and generic specification. That being said, the fact that the semantics is executable also provides a straightforward implementation, albeit not a terribly realistic one, performance-wise.

The most immediately obvious application is to simply compute the value of a contract. This is accomplished by the max-tropical semiring over real numbers, `Max Double`. By plugging the definition of `plus` and `times` for `Max` into Figure 3 it is easy to see that the `or` of two contracts is the maximum value of either contract, and that `both` simply sums the values. The worths of `expired` and `zero` are $-\infty$ and 0, respectively. (It is helpful to think of $-\infty$ as "undefined".) The remaining cases in Figure 3 are determined by the financial model, `static`:

```
static :: Currency -> Double -> Financial (Max Double) (Time -> Double)
```

This simplistic model assumes time-invariant exchange and interest rates. The full implementation of `static` can be found in the supplementary material.

Figure 4a shows the value at the epoch of progressively later ZCBs yielding €100 (with a fixed interest rate of 7% per time step). Note that more interest accrues if a contract is acquired longer before its expiry date, until the expiry date is reached and the value of the contract is $-\infty$. ρ is the derivative with respect to a 1% change in the interest rate, computed via automatic differentiation [1].

[9] For instance, their `truncate epoch zero` is not equal to our `expired`, because it can still be acquired at the epoch.

n	`(worth s7eur c`n` t0)`	ρ
0	$-\infty$	N/A
1	100	0
2	93	-0.87
3	86.49	-1.63
4	80.44	-2.29

(a) Value of `c`n at the epoch.

c	`(worth s0eur `c` t0)`
`both c1 c2’`	300
`c1 ‘and‘ c2’`	300
`both c0 c2’`	$-\infty$
`c0 ‘and‘ c2’`	200

(b) Compare the behaviour of `both` vs. `and`.

Fig. 4: Examples of contracts under the max-tropical semiring. Assume $n \in [0, 4]$, `t`n = `Finite` n, `c`n `= zcb t`n `100 EUR`, `c`n`’ = zcb t`n `200 EUR`, `s7eur = static EUR 0.07`, and `s0eur = static EUR 0.00`.

Figure 4b once more demonstrates the difference between `both` and `and` (the interest rate is held at 0% to make the difference more obvious). Since neither `c1` nor `c2’` has expired, both combinators behave the same, summing the values of both contracts. However, since `c0` has expired, `(both c0 c2’)` is $-\infty$, while `(c0 ‘and‘ c2’)` is equivalent to the remaining contract, `c2’`.

5 Conclusion

This paper has investigated the compositional nature of financial contracts from the perspective of abstract algebra, and equipped them with a semiring structure.

The advantages are threefold: First, semirings have straightforward axioms and properties, and admit a natural formulation of the denotational semantics as a semiring homomorphism. This theory is beneficial for equational reasoning, as is briefly demonstrated in the appendix, and it means that if the target domain satisfies the axioms, the semantics is correct, at least for the compositional part.

Second, the aim of defining a contract semiring leads directly to the `both` combinator, which seems slightly more powerful than the `and` combinator of Peyton Jones et al. [6], in the sense that it allows more contracts to be expressed in the language directly. To clarify this point, consider that it is always possible to define a contract that behaves like `(both c1 c2)` by using features from the host language (Haskell): simply trim the expiry dates of `c1` and `c2` to the earlier of the dates of `c1` and `c2`. Such tricks are not needed when `both` is a primitive, meaning that the intent of the contract is captured more precisely.

Third, formulating contracts in terms of semirings may reveal new applications inspired by semirings from the literature, such as those for automatic differentiation[10] [7, 2, 1], probabilities [3], polynomials, and Kleene-Algebras [5].

Acknowledgments. We are grateful for the helpful feedback of the anonymous reviewers. Part of this work was funded by FWO project 3E221387.

Disclosure of Interests. The authors have no competing interests.

[10] See the code repository for a gradient-based semantics.

Appendix: Deriving Expired

Let us derive the implementation of `expired` from the semantics:

```
  worth m k expired t
  [(W2): semiring homomorphism preserves nil]
= nil
  [(W1): nil = worth m k c t ⇐ t ≥ expiry c]
= worth m k (truncate t' c) t if t ≥ expiry (truncate t' c)
  [t' ≥ min (expiry c) t' = expiry (truncate t' c )]
= worth m k (truncate t' c) t if t ≥ t'
  [instantiate t' = epoch; t ≥ epoch]
= worth m k (truncate epoch c) t
  [instantiate c = zero]
= worth m k (truncate epoch zero) t
```

The choice of `zero` for `c` in the derivation is immaterial; any contract would do. Moreover, there are other forms of contracts that behave like `expired`. For instance, `(get zero)` is also `nil` everywhere, because the getting a contract with an infinite expiry date is ill-defined, i.e., `nil`. The definition above is preferable because it relies only on the non-finance specific part of the semantics.

References

1. van den Berg, B., Schrijvers, T., McKinna, J., Vandenbroucke, A.: Forward- or reverse-mode automatic differentiation: What's the difference? Sci. Comput. Program. **231**, 103010 (2024). https://doi.org/10.1016/J.SCICO.2023.103010, https://doi.org/10.1016/j.scico.2023.103010
2. Elliott, C.: The simple essence of automatic differentiation. Proc. ACM Program. Lang. **2**(ICFP), 70:1–70:29 (2018)
3. Erwig, M., Kollmansberger, S.: Functional pearls: Probabilistic functional programming in haskell. J. Funct. Program. **16**(1), 21–34 (2006)
4. Gibbons, J., Wu, N.: Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. p. 339–347. ICFP '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2628136.2628138, https://doi.org/10.1145/2628136.2628138
5. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. **19**(3), 427–443 (1997)
6. Peyton Jones, S.L., Eber, J., Seward, J.: Composing contracts: an adventure in financial engineering, functional pearl. In: ICFP. pp. 280–292. ACM (2000)
7. Rall, L.B.: Automatic Differentiation: Techniques and Applications, Lecture Notes in Computer Science, vol. 120. Springer (1981)
8. Yorgey, B.A.: Monoids: theme and variations *(functional pearl)*. In: Haskell. pp. 105–116. ACM (2012)