Rigged Contracts

Declarative Pearl

Alexander Vandenbroucke & Tom Schrijvers

Composing Contracts: An Adventure in Financial Engineering

Functional pearl

Simon Peyton Jones
Microsoft Research, Cambridge
simonpj@microsoft.com

Jean-Marc Eber
LexiFi Technologies, Paris
jeanmarc.eber@lexifi.com

Julian Seward
University of Glasgow
v-sewardj@microsoft.com

Abstract

Financial and insurance contracts do not sound like promising territory for functional programming and formal semantics, but in fact we have discovered that insights from programming languages bear directly on the complex subject of describing and valuing a large class of contracts.

We introduce a combinator library that allows us to describe such contracts precisely, and a compositional denotational semantics that says what such contracts are worth. We sketch an implementation of our combinator library in Haskell. Interestingly, lazy evaluation plays a crucial role.

1 Introduction

Consider the following financial contract, C: the right to choose on 30 June 2000 between

D_1 Both of:

At this point, any red-blooded functional programmer should start to foam at the mouth, yelling "build a combinator library". And indeed, that turns out to be not only possible, but tremendously beneficial.

The finance industry has an enormous vocabulary of jargon for typical combinations of financial contracts (swaps, futures, caps, floors, swaptions, spreads, straddles, captions, European options, American options, ...the list goes on). 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.

If, instead, we could define each of these contracts using a fixed, precisely-specified set of combinators, we would be in a much better position than having a fixed catalogue. For a start, it becomes much easier to *describe* new, unforeseen, contracts. Beyond that, we can systematically *analyse*, and *perform computations over* these new contracts, because they are described in terms of a fixed set of primitives.

The major thrust of this paper is to draw insights from the

Composing Contracts

The right to, on June 13th, 2024, choose between:
(a) the right to receive ¥260 on July 26th, 2024, and pay ¥270 on August 1st, 2024; or
(b) the right to receive ¥350 on July 26th, 2024, and pay ¥380 on August 1st, 2025.

x 100,000,000

What is this contract worth?

Contract Categories

1. Divide contracts into categories
2. Develop pricing models for each

image source: wikipedia

Contract Categories

1. Divide contracts into categories
2. Develop pricing models for each

American, European, Bermudan Options, Swap(tion)s, Futures, Forwards, Butterfly swaps, knock-in, knock-out, reverse knock-in, knock-out, ...

image source: wikipedia

Contract Categories

1. Divide contracts into categories
2. Develop pricing models for each

American, European, Bermudan Options, Swap(tion)s, Futures, Forwards, Butterfly swaps, knock-in, knock-out, reverse knock-in, knock-out, ...

Long exhaustive listing

image source: wikipedia

Contract Categories

1. Divide contracts into categories
2. Develop pricing models for each

American, European, Bermudan Options, Swap(tion)s, Futures, Forwards, Butterfly swaps, knock-in, knock-out, reverse knock-in, knock-out, ...

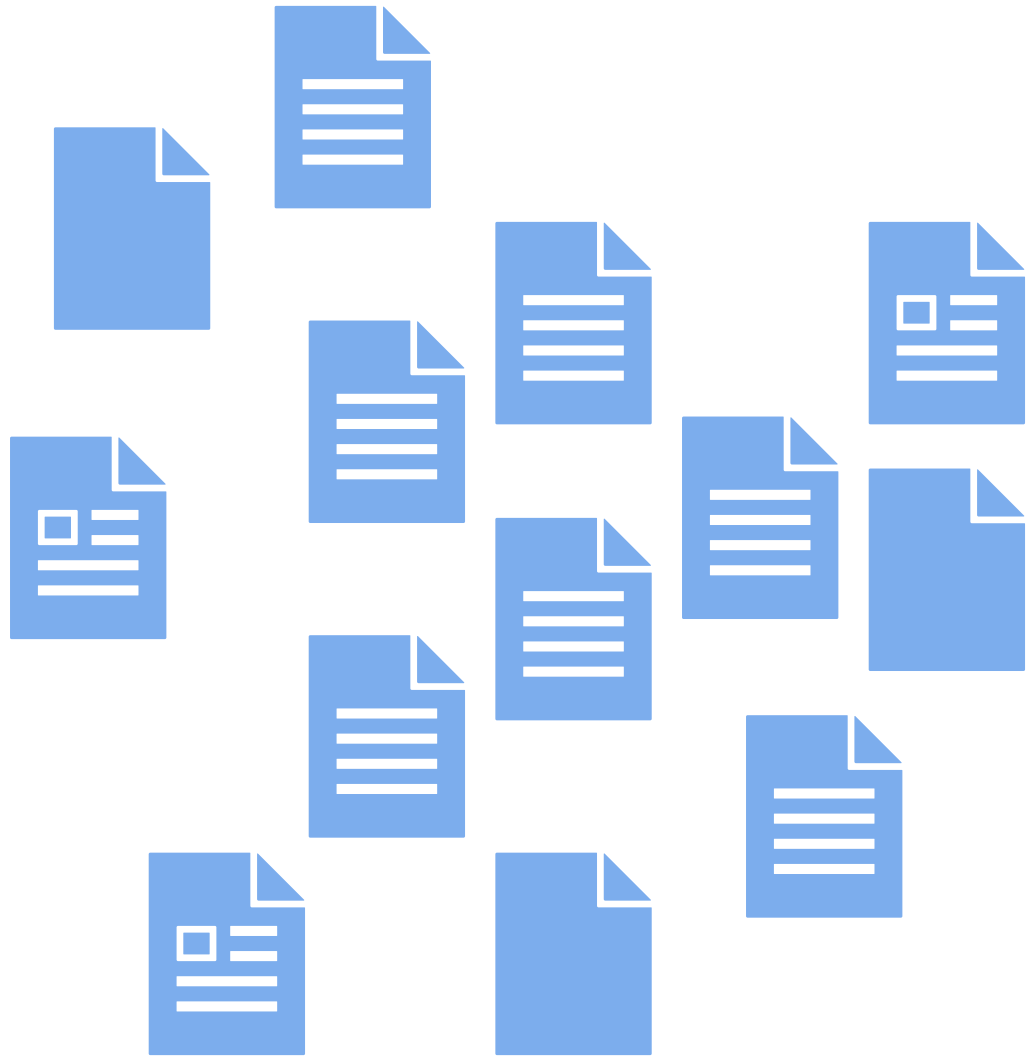

Long exhaustive listing


Still **incomplete**

image source: wikipedia

Contract Combinators

```
zero one give and or
truncate then scale
get anytime
```

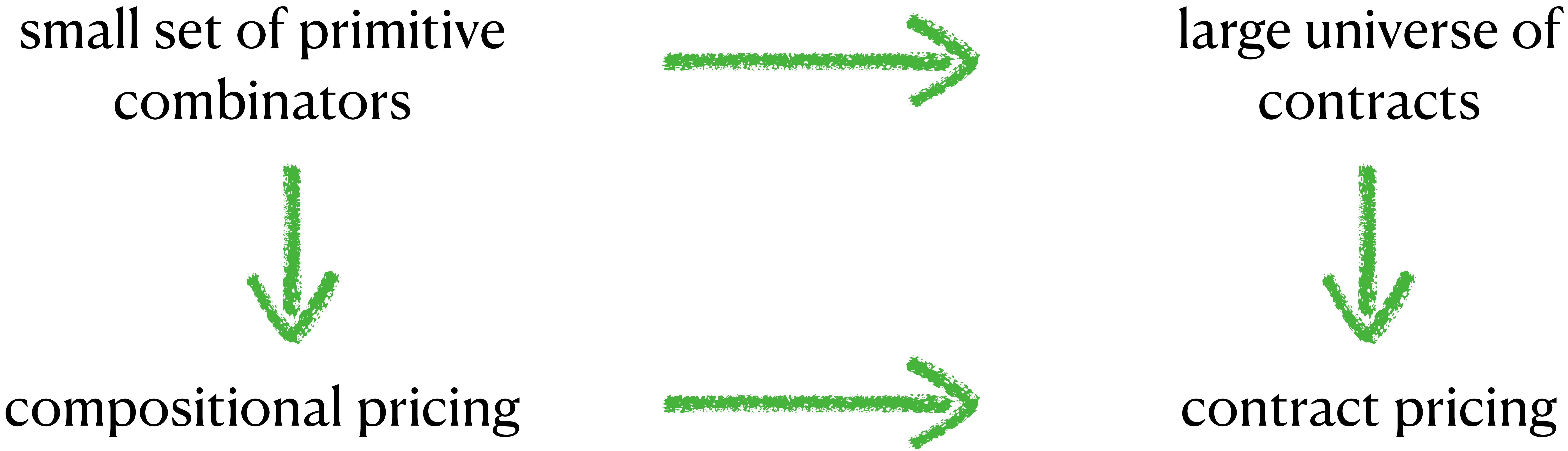


small set of primitive combinators

large universe of contracts

Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward. 2000. Composing contracts: an adventure in financial engineering, functional pearl. In ICFP. ACM, 280–292.

Contract Combinators



Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward. 2000. Composing contracts: an adventure in financial engineering, functional pearl. In ICFP. ACM, 280–292.

Rigged Contracts

The Zero-Coupon Bond

```
zcb :: Time → Double → Currency → Contract
```

```
zcb “21 May 2024” 100 JPY
```

A contract that pays ¥100 on the 21st of May 2024

The Zero-Coupon Bond

```
zcb :: Time → Double → Currency → Contract
zcb t a c = scaleK a $ get $ truncate (t + 1) $ one c
```

The Zero-Coupon Bond

```
zcb :: Time → Double → Currency → Contract
zcb t a c = scaleK a $ get $ truncate (t + 1) $ one c
```

receive one unit of c now

The Zero-Coupon Bond

```
zcb :: Time → Double → Currency → Contract
zcb t a c = scaleK a $ get $ truncate (t + 1) $ one c
```

receive one unit of c now

trim the **expiry date** to `(t+1)`

The Zero-Coupon Bond

```
zcb :: Time → Double → Currency → Contract
zcb t a c = scaleK a $ get $ truncate (t + 1) $ one c
```

receive one unit of c now

trim the **expiry date** to `(t+1)`

Horizon
latest point in time at which a contract can be acquired

⟷

Expiry date:
earliest point in time at which a contract can no longer be acquired

The Zero-Coupon Bond

```
zcb :: Time → Double → Currency → Contract
zcb t a c = scaleK a $ get $ truncate (t + 1) $ one c
```

receive one unit of c now

trim the **expiry date** to `(t+1)`

you *must* obtain the contract **right before** it expires

The Zero-Coupon Bond

```
zcb :: Time → Double → Currency → Contract
zcb t a c = scaleK a $ get $ truncate (t + 1) $ one c
```

receive one unit of c now

trim the **expiry date** to `(t+1)`

you *must* obtain the contract **right before** it expires

scale all rights and obligations by a constant amount

Both

```
zcb “10 May 2024” 155 JPY `both` give (zcb “10 Aug 2024” 100 USD)
```

1. acquire contracts on the left and right
2. before either has expired (i.e., “10 May 2024”)

Both

```
zcb “10 May 2024” 155 JPY `both` give (zcb “10 Aug 2024” 100 USD)
```

reverse all rights and obligations, you must pay 100 USD

1. acquire contracts on the left and right
2. before either has expired (i.e., “10 May 2024”)

Zero

```
zero `both` give (zcb "10 Aug 2024" 100 USD)
```

a contract that never expires and conveys neither rights nor obligations

```
= give (zcb "10 Aug 2024" 100 USD)
```

Zero

```
zero `both` give (zcb “10 Aug 2024” 100 USD)
```

a contract that never expires and conveys neither rights nor obligations

```
both zero c = c
```

Zero

```
zero `both` give (zcb "10 Aug 2024" 100 USD)
```

a contract that never expires and conveys neither rights nor obligations

```
both zero c = c
both (give c) c = truncate (expiry c) zero
```

Or

```
zcb "10 May 2024" 155 JPY `or` zcb "10 Aug 2024" 100 USD
```

1. acquire **exactly one** of the contracts on the left or right
2. before the respective contract has expired:
   after 10 May 2024 you can no longer pick the left contract

Expired

```
expired `or` zcb “10 Aug 2024” 100 USD
```

a contract that is always expired (expiry date = earliest possible date)

```
    = zcb “10 Aug 2024” 100 USD
```

Expired

```
expired `or` zcb “10 Aug 2024” 100 USD
```

a contract that is always expired (expiry date = earliest possible date)

```
or expired c = c
```

Expired

```
expired `or` zcb “10 Aug 2024” 100 USD
```

a contract that is always expired (expiry date = earliest possible date)

```
or expired c = c
```

```
both expired c = expired
```

Annihilation!

And

```
zcb “10 May 2024” 155 JPY `and` give (zcb “10 Aug 2024” 100 USD)
```

1. acquire contracts on the left and right
2. if either has expired that one can no longer be acquired

And

```
zcb “10 May 2024” 155 JPY `and` give (zcb “10 Aug 2024” 100 USD)
```

1.acquire contracts on the left and right
2.if either has expired that one can no longer be acquired

```
and c1 c2 = both c1 c2 `thereafter` or c1 c2
```

while the contract on the left is not expired, you acquire it, otherwise, you acquire the contract on the right

And

```
zcb “10 May 2024” 155 JPY `and` give (zcb “10 Aug 2024” 100 USD)
```

1.acquire contracts on the left and right
2.if either has expired that one can no longer be acquired

```
and c1 c2 = both c1 c2 `thereafter` or c1 c2
```

while c1 and c2 are both not expired, you acquire both, otherwise, you acquire the not yet expired contract

Less Ad-Hoc?

```
zero one give and or
truncate then scale
get anytime
```

Less Ad-Hoc?

```
zero one give and or
truncate then scale
get anytime

  -/+ truncate
```

Less Ad-Hoc?

```
zero one give and or
truncate then scale
get anytime
```

```
 -/+ truncate
   + expired
```

Less Ad-Hoc?

```
zero one give and or
truncate then scale
get anytime
```

```
 -/+ truncate
   + expired
 -/+ and
   + both
```

Less Ad-Hoc?

```
zero one give and or
truncate then scale
get anytime

  -/+ truncate
     + expired
  -/+ and
     + both
     - then
     + thereafter
```

Less Ad-Hoc?

```
zero one give and or
truncate then scale
get anytime
```

```
-/+ truncate
  + expired
-/+ and
  + both
  - then
  + thereafter
```

A. both and or are associative and commutative
B. zero and expired are resp. identities
C. expired annihilates both
D. both distributes over or

Semiring

These are the (Commutative) Semiring (Rig) Axioms!

Algebraic concepts are often useful when designing domain specific languages: monoids, monads, groups, ...,

Examples: numbers, (commutative) matrices, tropical semirings, derivatives, probabilities and expected values, booleans, Taylor models, ...

Pricing

Homomorphic Semantics

A mapping:

```
price : Contract → Semiring
```

preserving the semiring structure

Homomorphic Semantics

```
class Semiring r where
  nil   :: r             -- expired
  unit  :: r             -- zero
  plus  :: r -> r -> r   -- or
  times :: r -> r -> r   -- both

price expired      = nil
price zero         = unit
price (or   c1 c2) = price c1 `plus`  price c2
price (both c1 c2) = price c1 `times` price c2
```

Homomorphic Semantics

```haskell
class Semiring r => Multiplicative r where
  inv :: r -> r   -- give

price (give c) = inv (price c)
```

```
price c t | t ⩾ expiry c = nil

price Zero          t = unit
price (Both c1 c2) t = price c1 t `times` price c2 t
price (Or c1 c2)   t = price c1 t `plus` price c2 t
price (Give c)     t = inv (price c t)

price (Truncate t’ c)    t = price t c
price (Thereafter c1 c2) t | t < expiry c1 = price c1 t
                           | otherwise = price c2 t
price (One curr)  t = …
price (Get c)     t = …
price (Anytime c) t = …
```

Expiry

A mapping:

```
expiry : Contract → Time
```

preserving the semiring structure, where

$$Time = (\mathbb{N} \cup \{+\infty\}, 0, +\infty, \min, \max)$$

Pricing Semiring

Max Tropical

- `(price c t :: Max Double)` is the fair price at time t
- Captures original pricing semantics, but total!

```
instance Semiring (Max Double) where
 nil   = -∞
 unit  = 0
 plus  = max
 times = (+)
```

Pricing Semiring

$\frac{d}{dx}$ Gradient (Automatic Differentiation)

- `(price c t :: Gradient (Max Double))` is the fair price at time t and the derivative of the price with respect to one or more variables
- Derivatives are useful for optimisation, risk estimation, ...

And others ...

Conclusion

Conclusion

- ¥ More satisfactory, less ad-hoc combinators by realising contracts form a semiring
- € `Both` is slightly more powerful than `and`
- £ Formulating semantics as semiring morphisms points towards potential new applications