Composing contracts: an adventure in financial engineering

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

Jean-Marc Eber
Société Générale
jean-marc.eber@socgen.com

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

29th June 2000

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:

- D_{11} Receive £100 on 29 Jan 2001.
- D_{12} Pay £105 on 1 Feb 2002.

D_2 An option exercisable on 15 Dec 2000 to choose one of:

- D_{21} Both of:
  - D_{211} Receive £100 on 29 Jan 2001.
  - D_{212} Pay £106 on 1 Feb 2002.
- D_{22} Both of:
  - D_{221} Receive £100 on 29 Jan 2001.
  - D_{222} Pay £1012 on 1 Feb 2003.

The details of this contract are not important, but it is a simplified but realistic example of the sort of contract that is traded in financial derivative markets. What is important is that complex contracts, such as C, are formed by combining together simpler contracts, such as D_1, which in turn are formed from simpler contracts still, such as D_{11}, D_{12}.

To appear in the International Conference on Functional Programming, Montreal, Sept 2000

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 study of functional programming to illuminate the world of financial contracts. More specifically, our contributions are the following:

- We define a carefully-chosen set of combinators, and, through an extended sequence of examples in Haskell, we show that these combinators can indeed be used to describe a wide variety of contracts (Section 3).
- Our combinators can be used to *describe a contract*, but we also want to *process a contract*. Notably, we want to be able to *find the value of a contract*. In Section 4 we describe how to give an abstract *valuation semantics* to our combinators. A fundamentally-important property of this semantics is that it is *compositional*; that is, the value of a compound contract is given by combining the values of its sub-contracts.
- We sketch an implementation of our valuation semantics, using as an example a simple interest rate model and its associated lattice (Section 5). Lazy evaluation turns out to be tremendously important in translating the compositional semantics into a modular implementation (Section 5.3).

Stated in this way, our work sounds like a perfectly routine application of the idea of using a functional language

c, d, u	Contract
o	Observable
t, s	Date, time
k	Currency
x	Dimensionless real value
p	Value process
v	Random variable

Figure 1: Notational conventions

to define a domain-specific combinator library, thereby effectively creating an application-specific programming language. Such languages have been defined for parsers, music, animations, hardware circuits, and many others [15]. However, from the standpoint of financial engineers, our language is truly radical: they acknowledge that the lack of a precise way to describe complex contracts is "the bane of our lives" [1].

It has taken us a long time to boil down the immense soup of actively-traded contracts into a reasonably small set of combinators; but once that is done, new vistas open up, because a single formal description can drive all manner of automated processes. For example, we can generate schedules for back-office contract execution, perform risk analysis optimisations, present contracts in new graphical ways (e.g. decision trees), provide animated simulations, and so on.

This paper is addressed to a functional programming audience. We will introduce any financial jargon as we go.

2 Getting started

In this section we will informally introduce our notation for contracts, and show how we can build more complicated contracts out of simpler ones. We use the functional language Haskell [13] throughout.

2.1 A simple contract

Consider the following simple contract, known to the industry as *zero-coupon discount bond*: "receive £100 on 1st January 2010". We can specify this contract, which we name `c1`, thus:

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

Figure 1 summarises the notational conventions we use throughout the paper for variables, such as `c1` and `t1` in this definition.

The combinator `zcb` used in `c1`'s definition has the following type:

```
zcb :: Date -> Double -> Currency -> Contract
```

The first argument to `zcb` is a `Date`, which specifies a particular moment in time (i.e. both date and time). We provide a function, `date`, that converts a date expressed as a friendly character string to a `Date`.

```
date :: String -> Date
```

[1]The quote is from an informal response to a draft of our work

Now we can define our date `t1`:

```
t1,t2 :: Date
t1 = date "1530GMT 1 Jan 2010"
t2 = date "1200GMT 1 Feb 2010"
```

We will sometimes need to subtract dates, to get a time difference, and add a date and a time difference to get a new date.

```
type Days = Double     -- A time difference
diff :: Date -> Date -> Days
add  :: Date -> Days -> Date
```

We represent a time difference as a floating-point number in units of days (parts of days can be important).

2.2 Combining contracts

So `zcb` lets us build a simple contract. We can also combine contracts to make bigger contracts. A good example of such a combining form is `and`, whose type is:

```
and :: Contract -> Contract -> Contract
```

Using `and` we can define `c3`, a contract that involves two payments[2]:

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

That is, the holder of contract `c3` will benefit from a payment of £100 at time `t1`, and another payment of £200 at time `t2`.

In general, the contracts we can describe are between two parties, the *holder* of the contract, and the *counter-party*. Notwithstanding Biblical advice (Acts 10.35), by default the owner of a contract receives the payments, and makes the choices, specified in the contract. This situation can be reversed by the `give` combinator:

```
give :: Contract -> Contract
```

The contract `give c` is simply `c` with rights and obligations reversed, a statement we will make precise in Section 4.2. Indeed, when two parties agree on a contract, one acquires the contract `c`, and the other simultaneously acquires `(give c)`; each is the other's counter-party. For example, `c4` is a contract whose holder *receives* £100 at time `t1`, and *pays* £200 at time `t2`:

```
c4 = c1 `and` give c2
```

So far, each of our definitions has defined a new *contract* (`c1`, `c2`, etc). It is also easy to define a new *combinator* (a function that builds a contract). For example, we could define `andGive` thus:

```
andGive :: Contract -> Contract -> Contract
andGive c d = c `and` give d
```

Now we can give an alternative definition of `c4` (which we built earlier):

```
c4 = c1 `andGive` c2
```

[2]In Haskell, a function can be turned into an infix operator by enclosing it in back-quotes.

This ability to define new combinators, and *use them just as if they were built in*, is quite routine for functional programmers, but not for financial engineers.

3 Building contracts

We have now completed our informal introduction. In this section we will give the full set of primitives, and show how a wide variety of other contracts can be built using them. For reference, Figure 2 gives the primitive combinators over contracts; we will introduce these primitives as we need them.

3.1 Acquisition date and horizon

Figure 2 gives an English-language, but quite precise, description of each combinator. To do so, it uses two technical terms: acquisition date, and horizon. We begin by introducing them briefly.

Our language describes what a contract *is*. However, what the *consequences for the holder* of the contract depends on the date at which the contract is acquired, its *acquisition date*. (By "consequences for the holder" we mean the rights and obligations that the contract confers on the holder of a contract.) For example, the contract "receive £100 on 1 Jan 2000 and receive £100 on 1 Jan 2001" is worth a lot less if acquired after 1 Jan 2000, because any rights and obligations that fall due before the acquisition date are simply discarded.

The second fundamental concept is that of a contract's *horizon*, or *expiry date*: *the horizon, or expiry date, of a contract is the latest date at which it can be acquired.* A contract's horizon may be finite or infinite. The horizon of a contract is completely specified by the contract itself: given a contract, we can easily work out its horizon using the definitions in Figure 2. Note carefully, though, that a contract's rights and obligations may, in principle, extend well beyond its horizon. For example, consider the contract "the right to decide on or before 1 Jan 2001 whether to have contract C". This sort of contract is called an *option*. Its horizon is 1 Jan 2001 — it cannot be acquired after that date — but if one acquires it before then, the underlying contract C may (indeed, typically will) have consequences extending well beyond 1 Jan 2001.

To reiterate, the horizon of a contract is a property of the contract, while the acquisition date is not.

3.2 Discount bonds

Earlier, we described the zero-coupon discount bond: "receive £100 at time `t1`" (Section 2.1). At that time we assumed that `zcb` was a primitive combinator, but in fact it isn't. It is obtained by composing no fewer than four more primitive combinators. We begin with the `one` combinator:

```
c5 = one GBP
```

Figure 2 gives a careful, albeit informal, definition of `one`: if you acquire `(one GBP)`, you *immediately* receive £1. The contract has an infinite horizon; that is, there is no restriction on when you can acquire this contract.

But the bond we want pays £100 at `t1`, and no earlier, regardless of when the bond itself is acquired. To obtain this effect we use two other combinators, `get` and `truncate`, thus:

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

`truncate t c` is a contract that trims `c`'s horizon so that it cannot be acquired any later than `t`. `(get c)` is a contract that, when acquired, acquires the underlying contract `c` *at* `c`*'s horizon* — that is, at the last possible moment — regardless of when the composite contract `(get c)` is acquired. The combination of the two is exactly the effect we want, since the horizon of `(truncate t1 (one GBP))` is exactly `t1`. Like `one`, `get` and `truncate` are defined in Figure 2.

We are still not finished. The bond we want pays £100 not £1. We use the combinator `scaleK` to "scale up" the contract, thus:

```
c7 = scaleK 100 (get (truncate t1 (one GBP)))
```

We will define `scaleK` shortly, in Section 3.3. It has the type

```
scaleK :: Double -> Contract -> Contract
```

To acquire `(scaleK x c)` is to acquire `c`, but all the payments and receipts in `c` are multiplied by `x`. So we can, finally, define `zcb` correctly:

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

This definition of `zcb` effectively extends our repertoire of combinators, just as `andGive` did in Section 2.2, only more usefully. We will continually extend our library of combinators in this way.

Why did we go to the trouble of defining `zcb` in terms of four combinators, rather than making it primitive? Because it turns out that `scaleK`, `get`, `truncate`, and `one` are all independently useful. Each embodies a distinct piece of functionality, and by separating them we significantly simplify the semantics and enrich the algebra of contracts (Section 4). The combinators we present are the result of an extended, iterative process of refinement, leading to an interlocking set of decisions — programming language designers will be quite familiar with this process.

3.3 Observables and scaling

A real contract often mentions quantities that are to be measured on a particular date. For example, a contract might say "receive an amount in dollars equal to the noon Centigrade temperature in Los Angeles"; or "pay an amount in pounds sterling equal to the 3-month LIBOR spot rate[3] multiplied by 100". We use the term *observable* for an objective, *but perhaps time-varying*, quantity. By *"objective"* we mean that at any particular time the observable has a value that both parties to the contract will agree. The temperature in Los Angeles can be objectively measured; but the value to me of insuring my house is subjective, and is not an observable. Observables are thus a different "kind of thing" from contracts, so we give them a different type:

[3] The LIBOR spot rate is published daily in the financial press. For present purposes it does not matter what it means; all that matters is that it is an observable quantity.

`zero :: Contract`
: `zero` is a contract that may be acquired at any time. It has no rights and no obligations, and has an infinite horizon. (Section 3.4.)

`one :: Currency -> Contract`
: `(one k)` is a contract immediately pays the holder one unit of the currency `k`. The contract has an infinite horizon. (Section 3.2.)

`give :: Contract -> Contract`
: To acquire `(give c)` is to acquire all `c`'s rights as obligations, and vice versa. Note that for a bilateral contract `q` between parties A and B, A acquiring `q` implies that B acquires `(give q)`. (Section 2.2.)

`and :: Contract -> Contract -> Contract`
: If you acquire ``(c1 `and` c2)`` then you immediately acquire both `c1` (unless it has expired) and `c2` (unless it has expired). The composite contract expires when both `c1` and `c2` expire. (Section 2.2.)

`or :: Contract -> Contract -> Contract`
: If you acquire ``(c1 `or` c2)`` you must immediately acquire either `c1` or `c2` (but not both). If either has expired, that one cannot be chosen. When both have expired, the compound contract expires. (Section 3.4.)

`truncate :: Date -> Contract -> Contract`
: `(truncate t c)` is exactly like `c` except that it expires at the earlier of `t` and the horizon of `c`. Notice that `truncate` limits only the possible *acquisition date* of `c`; it does *not* truncate `c`'s rights and obligations, which may extend well beyond `t`. (Section 3.4.)

`then :: Contract -> Contract -> Contract`
: If you acquire ``(c1 `then` c2)`` and `c1` has not expired, then you acquire `c1`. If `c1` has expired, but `c2` has not, you acquire `c2`. The compound contract expires when both `c1` and `c2` expire. (Section 3.5.)

`scale :: Obs Double -> Contract -> Contract`
: If you acquire `(scale o c)`, then you acquire `c` at the same moment, except that all the rights and obligations of `c` are multiplied by the value of the observable `o` at the moment of acquisition. (Section 3.3.)

`get :: Contract -> Contract`
: If you acquire `(get c)` then you must acquire `c` at `c`'s expiry date. The compound contract expires at the same moment that `c` expires. (Section 3.2.)

`anytime :: Contract -> Contract`
: If you acquire `(anytime c)` you must acquire `c`, but you can do so at any time between the acquisition of `(anytime c)` and the expiry of `c`. The compound contract expires when `c` does. (Section 3.5.)

Figure 2: Primitives for defining contracts

```
noonTempInLA :: Obs Double
libor3m      :: Obs Double
```

In general, a value of type `Obs` d represents a time-varying quantity of type d.

In the previous section we used `scaleK` to scale a contract by a fixed quantity. The primitive combinator `scale` scales a contract by a time-varying value, that is, by an observable:

```
scale :: Obs Double -> Contract -> Contract
```

With the aid of `scale` we can define the (strange but realistic) contract "receive an amount in dollars equal to the noon Centigrade temperature in Los Angeles":

```
c8 = scale noonTempInLA (one USD)
```

Again, we have to be very precise in our definitions. Exactly when is the noon temperature in LA sampled? Answer (in Figure 2): when you acquire `(scale o c)` you immediately acquire `c`, scaling all the payments and receipts in `c` by the value of the observable `o` *sampled at the moment of acquisition.* So we sample the observable at a single, well-defined moment (the acquisition date) and then use that single number to scale the subsequent payments and receipts in `c`.

A very useful observable is one that has the same value at every time:

```
konst :: a -> Obs a
```

With its aid we can define `scaleK`:

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

Any arithmetic combination of observables is also an observable. For example, we may write:

```
ntLAinKelvin :: Obs Double
ntLAinKelvin = noonTempInLA + konst 373
```

We can use the addition operator, `(+)`, to add two observables, because observables are an instance of the `Num` class[4], which has operations for addition, subtraction, multiplication, and so on:

```
instance Num a => Num (Obs a)
```

(Readers who are unfamiliar with Haskell's type classes need not worry — all we need is that we can employ the usual arithmetic operators for observables.) These observables and their operations are, of course, reminiscent of Fran's *behaviours* [6]. Like Fran, we provide combinators for lifting functions to the observable level, `lift`, `lift2`, etc. Figure 3 gives the primitive combinators over observables.

3.4 European options

Much of the subtlety in financial contracts arises because the participants can exercise *choices*. We encapsulate choice in

[4] And indeed all the other numeric classes, such as `Real`, `Fractional`, etc

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

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

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

`instance Num a => Num (Obs a)`
: All numeric operations lift to the `Obs` type. The implementation is simple, using `lift` and `lift2`.

`time :: Date -> Obs Days`
: The value of the observable `(time t)` at time `s` is the number of days between `s` and `t`, positive if `s` is later than `t`.

There may be an arbitrary number of other primitive observables provided by a particular implementation. For example:

`libor :: Currency -> Days -> Days -> Obs Double`
: `(libor k m1 m2)` is an observable equal, at any time t, to the quoted forward (actuarial) rate in currency `k` over the time interval t `‘add‘ m1` to t `‘add‘ m2`.

Figure 3: Primitives over observables

two primitive combinators, `or` and `anytime`. The former allows one to choose *which* of two contracts to acquire (this section), while the latter allows one to choose *when* to acquire it (Section 3.5).

First, we consider the choice between two contracts:

```
or :: Contract -> Contract -> Contract
```

When you acquire the contract `(c1 ‘or‘ c2)`, you must immediately acquire either `c1` or `c2` (but not both). Clearly, `c1` can only be chosen at or before `c1`'s horizon, and similarly for `c2`. The horizon for `(c1 ‘or‘ c2)` is the latest of the horizons of `c1` and `c2`. Acquiring this composite contract, for example, after `c1`'s horizon but before `c2`'s horizon means that you can only "choose" to acquire contract `c2`. For example, the contract

```
zcb t1 100 GBP ‘or‘ zcb t2 110 GBP
```

gives the holder the right, if acquired before $min(\texttt{t1}, \texttt{t2})$, to choose immediately either to receive £100 at `t1`, or alternatively to receive £110 at `t2`.

A so-called *European option* gives the right to choose, at a particular date, whether or not to acquire an "underlying" contract:

```
european :: Date -> Contract -> Contract
```

For example, consider the contract `c5`:

```
c5 = european (date "24 Apr 2003") (
       zcb (date "12 May 2003") 0.4   GBP ‘and‘
       zcb (date "12 May 2004") 9.3   GBP ‘and‘
       zcb (date "12 May 2005") 109.3 GBP ‘and‘
       give (zcb (date "26 Apr 2003") 100 GBP)
   )
```

This contract gives the right to choose, on 24 Apr 2003, whether or not to acquire an underlying contract consisting of three receipts and one payment. In the financial industry, this kind of contract is indeed called a call on a coupon bond, giving the right, at a future date, to buy a bond for a prescribed price. As with `zcb`, we define `european` in terms of simpler elements:

```
european :: Date -> Contract -> Contract
european t u = get (truncate t (u ‘or‘ zero))
```

You can read this definition as follows:

- The primitive contract `zero` has no rights or obligations:

  ```
  zero :: Contract
  ```

- The contract `(u ‘or‘ zero)` expresses the choice between acquiring `u` and acquiring nothing.
- We trim the horizon of the contract `(u ‘or‘ zero)` to `t`, using the primitive combinator `truncate` (Figure 2).
- Finally, we use our `get` combinator to acquire it at that horizon.

We will repeatedly encounter the pattern `(truncate t (u ‘or‘ zero))`, so we will package it up into a new composite combinator:

```
perhaps :: Date -> Contract -> Contract
perhaps t u = truncate t (u ‘or‘ zero)
```

3.5 American options

The `or` combinator lets us choose *which* of two contracts to acquire. Let us now consider the choice of *when* to acquire a contract:

```
anytime :: Contract -> Contract
```

Acquiring the contract `anytime u` gives the right to acquire the "underlying" contract `u` at any time, from acquisition date of `anytime u` up to u's horizon. However, note that `u` *must* be acquired, albeit perhaps at the latest possible date.

An *American option* offers more flexibility than a European option. Typically, an American option confers the right to acquire an underlying contract *at any time between two dates*, or not to do so at all. Our first (incorrect) attempt to define such a contract might be to say:

```
american :: (Date,Date) -> Contract -> Contract
american (t1,t2) u    -- WRONG
  = anytime (perhaps t2 u)
```

but that is obviously wrong because it does not mention `t1`. We have to arrange that if we acquire the American contract before `t1` then the benefits are the same as if we acquired it at `t1`. So our next attempt is:

```
american (t1,t2) u    -- WRONG
  = get (truncate t1 (anytime (perhaps t2 u)))
```

But that is wrong too, because it does not allow us to acquire the American contract after `t1`. We really want to say "until `t1` you get this, and after `t1` you get that". We can express this using the `then` combinator:

```
american (t1,t2) u
  = get (truncate t1 opt) `then` opt
  where
    opt :: Contract
    opt = anytime (perhaps t2 u)
```

We give the intermediate contract `opt` an (arbitrary) name in a `where` clause, because we need to use it twice. The new combinator `then` is defined as follows: if you acquire the contract `` (c1 `then` c2) `` before `c1` expires then you acquire `c1`, otherwise you acquire `c2` (unless it too has expired).

3.6 Summary

We have now given the flavour of our approach to defining contracts. The combinators we have defined so far are not enough to describe all the contracts that are actively traded, and we are extend the set in ongoing work. However, our main conclusions are unaffected:

- Financial contracts can be described in a purely declarative way.
- A huge variety of contracts can be described in terms of a small number of combinators.

Identifying the "right" primitive combinators is quite a challenge. For example, it was a breakthrough to identify and separate the two forms of choice, `or` and `anytime`, and encapsulate those choices (and nothing else) in two combinators.

4 Valuation

We now have at our disposal a rich language for *describing* financial contracts. This is already useful for communicating between people — the industry lacks any such precise notation. But in addition, a precise description lends itself to automatic processing of various sorts. From a single contract description we may hope to generate legal paperwork, pictures, schedules and more besides. The most immediate question one might ask about a contract is, however, *what is it worth?* That is, *what would I pay to own the contract?* It is to this question that we now turn.

We will express contract valuation in two "layers":

Abstract evaluation semantics. First, we will show how to translate an arbitrary contract, written in our language, into a *value process*, together with a handful of operations over these processes. These processes correspond directly to the mathematical and stochastic machinery used by financial experts [14, 12].

Concrete implementation. A process is an abstract mathematical value. To make a computer calculate with processes we have to represent them somehow — this is the step from abstract semantics to concrete implementation. An implementation will consist of a financial model, associated to some discrete numerical method. A tremendous number of different financial *models* are used today; but only three families of *numerical methods* are widely used in industry: partial differential equations [17], Monte Carlo [1] and lattice methods [5].

This approach is strongly reminiscent of the way in which a compiler is typically structured. The program is first translated into a low-level but machine-independent intermediate language; many optimisations are applied at this level; and then the program is further translated into the instruction set for the desired processor (Pentium, Sparc, or whatever).

In a similar way, it is well worth separating these two levels in our financial setting: any theorems we can prove, or optimisations we can make, at the abstract level apply to all the concrete implementations.

Indeed, our abstract semantics serves as our reference model for what it means for two contracts to be the same. For example, here are two claims:

$$\begin{aligned}\texttt{get (get c)} &= \texttt{get c}\\ \texttt{give (c1 `or` c2)} &= \texttt{give c1 `or` give c2}\end{aligned}$$

In fact, the first is true, and the second is not, but how do we know for sure? Answer: we compare their valuation semantics, as we shall see in Section 4.6.

4.1 Value processes

Definition 1 (Value process.) *A* value process, *p, over type a, is a partial function from time to a random variable of type a. The random variable $p(t)$ describes the possible values for p at time t. We write the informal type definition*

$$\mathcal{PR}\ a = \mathcal{DATE} \hookrightarrow \mathcal{RV}\ a$$

(We use caligraphic font for types at the semantic level.) Because we need to work with different processes but defined on the same "underlying space" (filtration), such a value process is more precisely described as an *adapted stochastic process, given a filtration.* Such processes come equipped with a sophisticated mathematical theory [14, 12], but it is unlikely to be familiar to computer scientists, so we only present informal, intuitive notions. We usually abbreviate "value process" to simply "process". Be warned, though: "process" and "variable" mean quite different things to their conventional computer science meanings.

Both contracts and observables are modeled as processes. The underlying intuition is as follows:

- The value process for an observable `o` maps a time t to a random variable describing the possible values of `o` at t. For example, the value process for the observable "IBM stock price in US$" is a (total) function that maps a time to a real-valued random variable that describes the possible values of IBM's stock price in US$.
- The value process for a contract `c`, expressed in currency k is a (partial) function from a time, t, to a random variable describing the value, in currency k, of *acquiring* the contract `c` *at time* t.

$$
\begin{array}{rcl}
\mathcal{V}[\![]\!] : \texttt{Obs a} & \rightarrow & \mathcal{PR}\ \texttt{a} \\
\\
\mathcal{V}[\![\texttt{konst x}]\!] & = & \mathcal{K}(\texttt{x}) \\
\mathcal{V}[\![\texttt{time s}]\!] & = & \mathit{time}(\texttt{s}) \\
\mathcal{V}[\![\texttt{lift f o}]\!] & = & \mathit{lift}(\texttt{f}, \mathcal{V}[\![\texttt{o}]\!]) \\
\mathcal{V}[\![\texttt{lift2 f o1 o2}]\!] & = & \mathit{lift2}(\texttt{f}, \mathcal{V}[\![\texttt{o1}]\!], \mathcal{V}[\![\texttt{o2}]\!]) \\
\mathcal{V}[\![\texttt{libor k m1 m2}]\!] & = & \text{...omitted}
\end{array}
$$

Figure 5: Evaluation semantics for observables

These intuitions are essential to understand the rest of the paper.

A value process is, in general, a *partial* function of time; that is, it may not be defined for all values of its argument. Observables are defined for all time, and so do not need this flexibility; they define total processes. However, *contracts* are not defined for all time; the value process for a contract is undefined for times beyond its horizon.

4.2 From contracts to processes

How, then, are we to go from contracts and observables to processes? Figure 4 gives the complete translation from contracts to processes, while Figure 5 does the same for observables. These Figures do not look very impressive, but that is the whole point! Everything so far has been leading up to this point; our entire design is organised around the desire to give a simple, tractable, modular, valuation semantics. Let us look at Figure 4 in more detail.

The function $\mathcal{E}_k[\![]\!]$ takes a contract, `c`*, and maps it to a process describing, for each moment in time, the value in currency k of acquiring* `c` *at that moment.* For example, the equation for `give` (E1) says that the value process for `give c` is simply the negation of $\mathcal{E}_k[\![\texttt{c}]\!]$, the value process for `c`. Aha! What does "negation" mean? Clearly, we need not only the notion of a value process, but also a collection of operations over these processes. Negating a processes is one such operation; the negation of a process p is simply a function that maps each time, t, to the negation of $p(t)$. It is an absolutely straightforward exercise to "lift" all operations on real numbers to operate point-wise on processes. (This, in turn, requires us to negate a random variable, but doing so is simple.) We will need a number of other operations over processes. They are summarised in Figure 6, but we will introduce each one as we need it.

Next, consider equation (E2). The `and` of two contracts is modeled by taking the sum of their two value processes; we need three equations to give the value of $\mathcal{E}_k[\![]\!]$ when t is earlier than the horizon of both contracts, when it is earlier than one but later than the other, and vice versa. In the fourth case — i.e. for times beyond both horizons — the evaluation function is simply undefined. We use the notation "$on\{t \mid \ldots t \ldots\}$" to indicate that the corresponding equation applies for only part of the (time) domain of $\mathcal{E}_k[\![\texttt{c}]\!]$.

Figure 7 specifies formally how to calculate the horizon $\mathrm{H}(\texttt{c})$ of a contract `c`. It returns ∞ as the horizon of a contract with an infinite horizon; we extend $\leq$ and max in the obvious way to such infinities.

These primitives are independent of the evaluation model

$\mathcal{K} : a \rightarrow \mathcal{PR}\ a$
: The process $\mathcal{K}(x)$ is defined at all times to have value x.

$time : \mathcal{DATE} \rightarrow \mathcal{PR}\ \mathbf{R}$
: The process $time(s)$ is defined at all times t to be the number of days between s and t. It is positive if t is later than s.

$lift : (a \rightarrow b) \rightarrow \mathcal{PR}\ a \rightarrow \mathcal{PR}\ b$
: Apply the specified function to the argument process point-wise. The result is defined only where the arguments process is defined.

$lift2 : (a \rightarrow b \rightarrow c) \rightarrow \mathcal{PR}\ a \rightarrow \mathcal{PR}\ b \rightarrow \mathcal{PR}\ c$
: Combine the two argument processes point-wise with the specified function. The result is defined only where both arguments are defined.

These primitives are dependent on the particular model

$disc_k^T : \mathcal{RV}_T\ \mathbf{R} \rightarrow \mathcal{PR}\ \mathbf{R}$
: The primitive $disc_k^T$ maps a real-valued random variable at date T, expressed in currency k, to its "fair" equivalent stochastic value process in the same currency k.

$exch_{k1}(k2) : \mathcal{PR}\ \mathbf{R}$
: $exch_{k1}(k2)$ is a real-valued process representing the value of one unit of $k2$, expressed in currency $k1$. This is simply the process representing the quoted exchange rate between the currencies.

$snell_k^T : \mathcal{PR}\ \mathbf{R} \rightarrow \mathcal{PR}\ \mathbf{R}$
: The primitive $snell_k^T$ calculates the Snell envelope of its argument. It uses the probability measure associated with the currency k.

Figure 6: Model primitives

Equation (E3) does the same for the `or` combinator. Again, by design, the combinator maps to a simple mathematical operation, max. One might wonder why we defined a value process to be a partial function, rather than a total function that is zero beyond its horizon. Equation (E3) gives the answer: beyond `c1`'s horizon one is *forced* to choose `c2`. In general, $max(v_1, 0) \neq v_1$!

Equation (E4) is nice and simple. To scale a contract `c` by a time-varying observable `o`, we simply multiply the value process for the contract $\mathcal{E}_k[\![\texttt{c}]\!]$ by the value process for the observable — remember that we are modeling each observable by a value process. We express the latter as $\mathcal{V}[\![\texttt{o}]\!]$, defined in Figure 5 in a very similar fashion to $\mathcal{E}_k[\![]\!]$. At first this seems odd: how can we scale point-wise, when the scaling applies to *future* payments and receipts in `c`? Recall that the value process for `c` at a time t gives the value of acquiring `c` at t. Well, if this value is v then the value of acquiring the same contract with all payments and receipts scaled by x is certainly $v * x$. Our definition of `scale` in Figure 2 was in fact

$$
\begin{array}{llcll}
 & \mathcal{E}_k[\![]\!] : \texttt{Contract} & \rightarrow & \mathcal{PR}\ \mathbf{R} & \\
(E1) & \mathcal{E}_k[\![\texttt{give c}]\!] & = & -\mathcal{E}_k[\![\texttt{c}]\!] & \\
(E2) & \mathcal{E}_k[\![\texttt{c1 `and` c2}]\!] & = & \mathcal{E}_k[\![\texttt{c1}]\!] + \mathcal{E}_k[\![\texttt{c2}]\!] & \text{on } \{t \mid t \leq H(\texttt{c1}) \ \wedge\ t \leq H(\texttt{c2})\} \\
 & & & \mathcal{E}_k[\![\texttt{c1}]\!] & \text{on } \{t \mid t \leq H(\texttt{c1}) \ \wedge\ t > H(\texttt{c2})\} \\
 & & & \mathcal{E}_k[\![\texttt{c2}]\!] & \text{on } \{t \mid t > H(\texttt{c1}) \ \wedge\ t \leq H(\texttt{c2})\} \\
(E3) & \mathcal{E}_k[\![\texttt{c1 `or` c2}]\!] & = & max(\mathcal{E}_k[\![\texttt{c1}]\!], \mathcal{E}_k[\![\texttt{c2}]\!]) & \text{on } \{t \mid t \leq H(\texttt{c1}) \ \wedge\ t \leq H(\texttt{c2})\} \\
 & & & \mathcal{E}_k[\![\texttt{c1}]\!] & \text{on } \{t \mid t \leq H(\texttt{c1}) \ \wedge\ t > H(\texttt{c2})\} \\
 & & & \mathcal{E}_k[\![\texttt{c2}]\!] & \text{on } \{t \mid t > H(\texttt{c1}) \ \wedge\ t \leq H(\texttt{c2})\} \\
(E4) & \mathcal{E}_k[\![\texttt{o `scale` c}]\!] & = & \mathcal{V}[\![\texttt{o}]\!] * \mathcal{E}_k[\![\texttt{c}]\!] & \\
(E5) & \mathcal{E}_k[\![\texttt{zero}]\!] & = & \mathcal{K}0 & \\
(E6) & \mathcal{E}_k[\![\texttt{truncate T c}]\!] & = & \mathcal{E}_k[\![\texttt{c}]\!] & \text{on } \{t \mid t \leq \texttt{T}\} \\
(E7) & \mathcal{E}_k[\![\texttt{c1 `then` c2}]\!] & = & \mathcal{E}_k[\![\texttt{c1}]\!] & \text{on } \{t \mid t \leq H(\texttt{c1})\} \\
 & & & \mathcal{E}_k[\![\texttt{c2}]\!] & \text{on } \{t \mid t > H(\texttt{c1})\} \\
(E8) & \mathcal{E}_k[\![\texttt{one k2}]\!] & = & exch_k(\texttt{k2}) & \\
(E9) & \mathcal{E}_k[\![\texttt{get c}]\!] & = & disc_k^{H(\texttt{c})}(\mathcal{E}_k[\![\texttt{c}]\!](H(c))) & \text{if } H(\texttt{c}) \neq \infty \\
(E10) & \mathcal{E}_k[\![\texttt{anytime c}]\!] & = & snell_k^{H(\texttt{c})}(\mathcal{E}_k[\![\texttt{c}]\!]) & \text{if } H(\texttt{c}) \neq \infty
\end{array}
$$

Figure 4: Compositional evaluation semantics for contracts

$$
\begin{array}{rcl}
H(\texttt{zero}) & = & \infty \\
H(\texttt{one k}) & = & \infty \\
H(\texttt{c1 `and` c2}) & = & max(H(\texttt{c1}), H(\texttt{c2})) \\
H(\texttt{c1 `or` c2}) & = & max(H(\texttt{c1}), H(\texttt{c2})) \\
H(\texttt{c1 `then` c2}) & = & max(H(\texttt{c1}), H(\texttt{c2})) \\
H(\texttt{truncate t c}) & = & min(\texttt{t}, H(\texttt{c})) \\
H(\texttt{scale o c}) & = & H(\texttt{c}) \\
H(\texttt{anytime c}) & = & H(\texttt{c}) \\
H(\texttt{get c}) & = & H(\texttt{c})
\end{array}
$$

Figure 7: Definition of horizon

driven directly by our desire to express its semantics in a simple way. Simple semantics gives rise to simple algebraic properties (Section 4.6).

The equations for `zero`, `truncate`, and `then` are also easy. Equation (E5) delivers the constant zero process, while Equation (E6) truncates a process simply by limiting its domain — remember, again, that the time argument of a process models the acquisition date. Equation (E7) behaves like the first process in its domain, and elsewhere like the second.

4.3 Exchange rates

The top group of operations over value processes defined in Figure 6 are generic – they are unrelated to a particular financial model. But we can't get away with that for ever. The lower group of primitives in the same figure are specific to financial contracts, and they are used in the remaining equations of Figure 4.

Consider equation (E8) in Figure 4. It says that to get the value process for one unit of currency `k2`, expressed in currency `k`, is simply the exchange-rate process between `k2` and `k` namely $exch_{\texttt{k}}(\texttt{k2})$ (Figure 6). Where do we get these exchange-rate processes from? When we come to implementation, we will need some (numerical) assumption about future evolution of exchange rates, but for now it suffices to treat the exchange rate processes as primitives. However, there are important relationships between them! Notably:

$$
\begin{array}{lrcl}
(A1) & exch_k(k) & = & \mathcal{K}(1) \\
(A2) & exch_{k_2}(k_1) * exch_{k_3}(k_2) & = & exch_{k_3}(k_1)
\end{array}
$$

That is, exchange-rate process between a currency and itself is everywhere unity; and it makes no difference whether we convert k_1 directly into k_3 or whether we go via some intermediate currency k_2.

4.4 Interest rates

Next, consider equation (E9). The `get` combinator acquires the underlying contract `c` *at its horizon*, $H(\texttt{c})$. (`get c` is undefined if `c` has an infinite horizon.) It does not matter what `c`'s value might be at earlier times; all that matters is `c`'s value at its horizon, which is described by the random variable $\mathcal{E}_k[\![\texttt{c}]\!](H(\texttt{c}))$. What is the value of `get c` at earlier times? To answer that question we need a specification of future evolution of interest rates, that is an interest rate model.

Let's consider a concrete example:

```
c = get (scaleK 10 (truncate t (one GBP)))
```

where `t` is one year from today. The underlying contract `(scaleK 10 (truncate t (one GBP)))` pays out £10 immediately it is acquired; the `get` acquires it at its horizon, namely `t`. So the value of `c` at `t` is just £10. Before `t`, though, it is not worth as much. If I expect interest rates to average[5] (say) 10% over the next year, a fair price for `c` today would be about £9.

Just as the primitive *exch* encapsulates assumptions about future exchange rate evolution, so the primitive *disc* en-

[5]For the associated *risk-neutral* probability, but we will not go in these financial details here.

capsulates an interest rate evolution (Figure 6). It maps a *random variable* describing a payout, in a particular currency, at a particular date, into a *process* describing the value of that payout at earlier dates, in the same currency. Like *exch*, there are some properties that any no-arbitrage financial model should satisfy. Notably:

$$
\begin{array}{lrcl}
(A3) & disc_k^t(v)(t) & = & v \\
(A4) & exch_{k_1}(k_2) * disc_{k_2}^t(v) & = & disc_{k_1}^t(exch_{k_1}(k_2)(t) * v) \\
(A5) & disc_k^t(v_1 + v_2) & = & disc_k^t(v_1) + disc_k^t(v_2)
\end{array}
$$

The first equation says that *disc* should be the identity at its horizon; the second says that the interest rate evolution of different currencies should be compatible with the assumption of evolution of exchange rates. The third[6] is often used in a right-to-left direction as optimisations: rather than perform discounting on two random variables separately, and then add the resulting process trees, it is faster to add the random variables (a single column) and then discount the result. Just as in an optimising compiler, we may use identities like these to transform (the meaning of) our contract into a form that is faster to execute.

One has to be careful, though. Here is a plausible property that does *not* hold:

$$disc_k^t(max(v_1, v_2)) = max(disc_k^t(v_1), disc_k^t(v_2))$$

It is plausible because it would hold if v_1, v_2 were single numbers and *disc* were a simple multiplicative factor. But v_1 and v_2 are random variables, and the property is false.

Equation (E10) uses the *snell* operator to give the meaning of `anytime`. This operator is mathematically very subtle, but it has a simple characterisation: $snell_k^t(p)$ is the smallest process q such that

- $q \geq p$. Since we can exercise the option at any time, `anytime c` at all times better than `c`.
- $\forall t. q \geq disc_k^t(q(t))$. Since we can always defer exercising the option, `(anytime c)` is always better than the same contract acquired later.

4.5 Observables

We can only value contracts over observables that we can model. For example, we can only value a contract involving the temperature in Los Angeles if we have a model of the temperature in Los Angeles. Some such observables clearly require separate models. Others, such as the LIBOR rate and the price of futures, can incestuously be modeled as the value of particular contracts. We omit all the details here; Figure 5 gives the semantics only for the simplest observables. This is not unrealistic, however. One can write a large range of contracts with our contract combinators and only these simple observables.

4.6 Reasoning about contracts

Now we are ready to use our semantics to answer the questions we posed at the beginning of Section 4. First, is this equation valid?

[6] The financially educated reader should note that we assume here implicitly what is called *complete* markets.

```
get (get c) = get c
```

We take the meaning of the left hand side in some arbitrary currency k:

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathcal{E}_k[\![\texttt{get (get c)}]\!]} & \\
= & disc_k^{h_1}(\mathcal{E}_k[\![\texttt{get c}]\!](h_1)) & \text{by (E9)} \\
= & disc_k^{h_1}(disc_k^{h_2}(\mathcal{E}_k[\![\texttt{c}]\!](h_2))(h_1)) & \text{by (E9)} \\
= & disc_k^{h_2}(disc_k^{h_2}(\mathcal{E}_k[\![\texttt{c}]\!](h_2))(h_2)) & \text{since } h_1 = h_2 \\
= & disc_k^{h_2}(\mathcal{E}_k[\![\texttt{c}]\!](h_2)) & \text{by (A3)} \\
= & \mathcal{E}_k[\![\texttt{get c}]\!] & \text{by (E9)} \\
\multicolumn{3}{l}{where} \\
 & h_1 = H(\texttt{get c}) & \\
 & h_2 = H(\texttt{c}) &
\end{array}
$$

In a similar way, we can argue this plausible equation is false:

```
give (c1 `or` c2) = give c1 `or` give c2
```

The proof is routine, but its core is the observation that

$$-max(a, b) \neq max(-a, -b)$$

The `or` operation takes the maximum, while `give` performs negation, and negation simply does not distribute over maximum.

Our combinators satisfy a rich set of equalities, such as that given for `get` above. Some of these equalities have side conditions; for example:

```
scale o (c1 `or` c2) = scale o c1 `or` scale o c2
```

holds only if $\texttt{o} \geq 0$, for exactly the same reason that `get` does not commute with `or`. Hang on! What does it mean to say that "$\texttt{o} \geq 0$"? We mean that `o` is positive for all time. More generally, as well as equalities between contracts, we have also developed a notion of ordering between both observables and contracts, $\texttt{c1} \geq \texttt{c2}$, pronounced *"*`c1` *dominates* `c2`*"*.

Equalities, such as the ones given above, can be used as optimising transformations in a valuation engine. A "contract compiler" can use these identies to transform a contract, expressed in the intermediate language of value processes (see the introduction to Section 4), into a form that can be valued more cheaply.

4.7 Summary

This completes our description of the abstract evaluation semantics. From a programming-language point of view, everything is quite routine, including our proofs. But we stress that it is most unusual to find formal proofs in the finance industry at this level of abstraction. We have named and tamed the complicated primitives (*disc*, *exch*, etc): the laws they must satisfy give us a way to prove identities about contracts without having to understand much about random variables. The mathematical details are arcane, believe us!

5 Implementation

Our evaluation semantics is not only an abstract beast. We can also regard Figures 4 and 5 as a translation from our contract language into a lower-level language of processes, whose combinators are the primitives of Figure 6. Then we

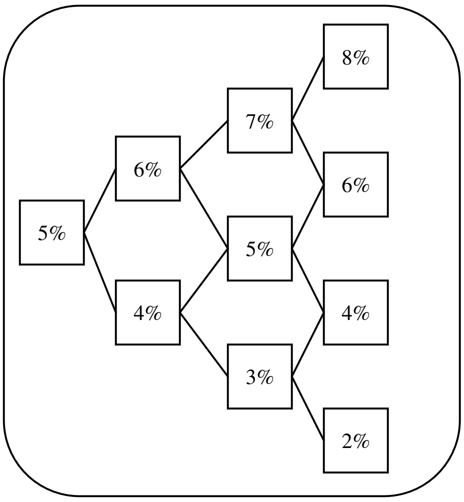

Figure 8: A short term interest rate evolution

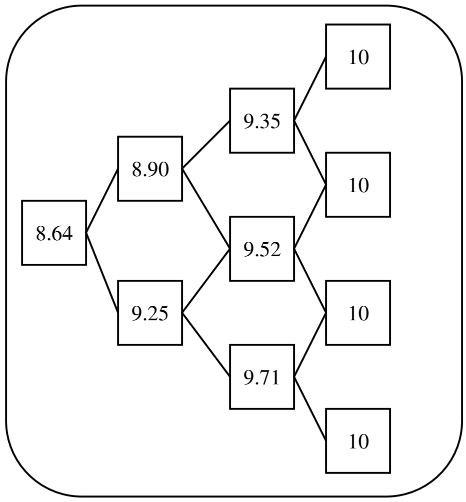

Figure 9: A Ho and Lee valuation lattice

can optimise the process-level description, using (A1)-(A5). Finally, all (ha!) we need to do is to implement the process-level primitives, and we will be able to value an arbitrary contract.

The key decision is, of course, how we implement a value process. A value process has to represent *uncertainty about the future* in an explicit way. There are numerous ways to model this uncertainty. Rather than try to be general, we will simply pick the Ho and Lee model, and use a lattice method to evaluate contracts with it [8].

5.1 An interest rate model

In the typical Ho and Lee numerical scheme, the interest rate evolution is represented by a lattice (or "recombining tree"), as depicted in Figure 8. Each column of the tree represents a discrete time step, and time increases from left to right. Time zero represents "now". As usual with discrete models, there is an issue of how long a time step will be; we won't discuss that further here, but we note in passing that the time steps need not be of uniform size.

At each node of the tree is associated a one period short term interest rate, shortly denominated the interest rate from now on. We know today's interest rate, so the first column in the tree has just one element. However, there is some uncertainty of what interest rates will evolve to by the end of the first time step. This is expressed by having two interest-rate values in the second column; the idea is that the interest rate will evolve to one of these two values with equal probability. In the third time step, the rates split again, *but the down/up path joins the up/down path,* so there are only three rates in the third column, not four. This is why the structure is called a lattice; it makes the whole scheme computationally feasible by giving only a linear growth in the width of the tree with time. Of course, the tree is only a discrete approximation of a continuous process; its recombining nature is just a choice for efficiency reasons. We write R_t for the vector of rates in time-step t, and $R_{t,i}$ for the i'th member of that vector, starting with 0 at the bottom. Thus, for example, $R_{2,1} = 5\%$. The actual numbers in Figure 8 are unrealistically regular: in more elaborated interest rate models, they will not be evenly spaced but only monotonically distributed in each column.

5.2 Value processes

So much for the interest rate model. A value process is modeled by a lattice of exactly the same shape as the interest rate evolution, except that we have a *value* at each node instead of an *interest rate*. Figure 9 shows the value process tree for our favourite zero-coupon bond

```
c7 = get (scaleK 10 (truncate t (one GBP)))
```

evaluated in pounds sterling (GBP). Using our evaluation semantics we have

$$\mathcal{E}_{GBP}[\![\texttt{c7}]\!] = disc^{\texttt{t}}_{\texttt{GBP}}(\mathcal{K}(10)(\texttt{t}))$$

In the Figure, we assume that the time `t` is time step 3. At step 3, therefore, the value of the contract `c` is certainly 10 at all nodes, because `c` unconditionally delivers £10 at that time — remember axiom (A3). At time step 2, however, we must discount the £10 by the interest rate appropriate to that time step. We compute the value at each node of time-step 2 by averaging the two values in its successors, and then discounting the average value back one time step using the interest rate associated to that node[7]. Using the same notation for the value tree V as we used for the rate model R, we get the equation:

$$V_{t,i} = \frac{V_{t+1,i} + V_{t+1,i+1}}{2(1 + R_{t,i}\Delta t)}$$

where Δt is the size of the time step. Using this equation we can fill in the rest of the values in the tree, as we have done in Figure 9. The value in time step 0 is the current value of the contract, in pounds sterling. i.e £8.64.

[7] For evident presentation reasons, we don't care about the fact that the Ho and Lee model is member of a class of models that admit in fact a *closed-form solution* for zero-coupon bonds.

In short, a lattice implementation works as follows:

- A value process is represented by a lattice, in which each column is a discrete representation of a random variable. The value in each node is one of the possible values the variable can take, and in our very simple setting the number of paths from the root to the node is proportional to the probability that the variable will take that value. We will say a bit more about how to represent such a tree in the next subsection.
- The generic operations, in the top half of Figure 6, are easy to implement. $\mathcal{K}(x)$ is a value process that is everywhere equal to x. $time(t)$ is a process in which the values in a particular column are all equal to the number of days between that column's time and t. $lift(f,p)$ applies f to p point-wise; $lift2(f,p_1,p_2)$ "zips together" p_1 and p_2, combining corresponding values point-wise with f.
- The model-specific operations of Figure 6 are a bit harder. We have described how to implement *disc*, which uses the interest rate model. *exch* is actually rather easier (multiply the value process point-wise by a process representing the exchange-rate). The *snell* primitive takes a bit more work, and we do not describe it in detail here. Roughly speaking, a possible implementation may be: take the final column of the tree, discount it back one time step, take the maximum of that column with the corresponding column of the original tree, and then repeat that process all the way back to the root.

The remaining high-level question is: in the (big) set of possible interest rate models, what is a "good" model? The answer is rather incestuous. A candidate interest rate model should price correctly those contracts that are widely traded: one can simply look up the current market prices for them, and compare them with the calculated results. So we look for and later adjust the interest rate model until it fits the market data for these simple contracts. Now we are ready to use the model to compute prices for more exotic contracts. The entire market is a gigantic feedback system, and active research studies the problem of its stability.

5.3 Implementation in Haskell

We have two partial implementations of (earlier versions of) these ideas, one of which is implemented as a Haskell combinator library. The type `Contract` is implemented as an algebraic data type, with one constructor for each primitive combinator:

```
data Contract = One Date Currency
              | Give Contract
              | ...
```

The translation to processes is done by a straightforward recursive Haskell implementation of $\mathcal{E}_k[\![]\!]$:

```
eval :: Model -> Currency -> Contract -> ValProc
```

Here, `Model` contains the interest rate evolutions, exchange rate evolutions, and whatever other "underlyings" are necessary to evaluate the contract.

Our first implementation used the following representation for a value process:

```
type ValProc = (TimeStep, [Slice])
type Slice   = [Double]
```

A value process is represented by a pair of (a) the process's horizon, and (b) a list of slices (or columns), one per time step *in reverse time order.* The first slice is at the horizon of the process, the next slice is one time step earlier, and so on. Since the (fundamental) discount recurrence equation (Section 5.1) works backwards in time, it is convenient to represent the list this way round. Each slice is one element shorter than the one before.

Laziness plays a very important role, for two reasons:

- *Process trees can become very large*, since their size is quadratic in the number of time steps they cover. A complex contract will be represented by combining together many value trees; it would be Very Bad to fully evaluate these sub-trees, and only then combine them. Lazy evaluation automatically "pipelines" the evaluation algorithm, so that only the "current slice" of each value tree is required at any one moment.
- *Only part of a process tree may be required.* Consider again our example contract

  ```
  c = get (scaleK 10 (truncate t (one GBP)))
  ```

  The value process for `(scaleK 10 (truncate t (one GBP)))` is a complete value process, all the way back to time-step zero, with value 10 everywhere. But `get` samples this value process only at its horizon — there is no point in computing its value at any earlier time. By representing a value process as a lazily-evaluated list we get the "right" behaviour automatically.

Microsoft Research collaborates closely with Lombard Risk Systems Ltd, who have a production tree-based valuation system in C++. It uses a clever but complex event-driven engine in which a value tree is represented by a single slice that is mutated as time progresses. There is never a notion of a complete tree. The Haskell implementation treats trees as first class values, and this point of view offers a radical new perspective on the whole evaluation process. We are hopeful that some of the insights from our Haskell implementation may serve to inform and improve the efficient C++ implementation.

The Haskell version takes around 900 lines of Haskell to support a working, albeit limited, contract valuation engine, complete with a COM interface [7] that lets it be plugged into Lombard's framework. It is not nearly as fast as the production code, but it is not unbearably slow either — for example, it takes around 20 seconds to compute the value of a contract with 15 sub-contracts, over 500 time steps, on a standard desktop PC. Though it lacks much functionality, the compositional approach means that can already value some contracts, such as options over options, that the production system cannot. (The production system is not fundamentally incapable of such feats; but it is programmed on a case-by-case basis, and the more complicated cases are dauntingly hard to implement.)

5.4 Memoisation

In functional programming terms, most of this is quite straightforward. There is a nasty practical problem, however, that repeatedly bites people who embed a domain specific language in a functional language. Consider the contract

```
c10 = join `and` join
    where
      join = <stuff> `or` <more stuff>
```

Here, `join` is a shared sub-contract of `c10` much like `opt` in our definition of `american` (Section 3.5). The trouble is that `eval` will evaluate the two branches of the `and` at the root of `c10`, oblivious of the fact that these two branches are the same. In fact, `eval` will do all the work of evaluating `join` twice! There is no way for `eval` to tell that it has "seen this argument before".

This problem arises, in various guises, in almost every embedded domain-specific language. We have seen it in Fran's reactive animations [6], the difficulty of extracting net-lists from Hawk circuit descriptions [4], and in other settings besides. What makes it particularly frustrating is that the sharing is absolutely apparent in the source program.

One "solution" is to suggest that `eval` be made a memo function [10, 3, 11], but we do not find it satisfactory. Losing sharing can give rise to an unbounded amount of duplicated work, so it seems unpleasant to relegate the maintenance of proper sharing to an operational mechanism. For example, a memo function may be deceived by unevaluated arguments, or automatically-purged memo tables, or whatever. For now we simply identify it as an important open problem that deserves further study. The only paper that addresses this issue head on is [2]: it proposes one way to make sharing observable, but leaves open the question of memo functions.

6 Putting our work in context

At first sight, financial contracts and functional programming do not have much to do with each other. It has been a surprise and delight to discover that many of the insights useful in the design, semantics, and implementation of programming languages can be applied directly to the description and evaluation of contracts. One of us (Eber) has been developing this idea for nearly ten years at Société Générale. The others (Peyton Jones and Seward) came to it much more recently, through a fruitful partnership with Lombard Risk Systems Ltd. The original idea was to apply functional programming to a realistic problem, and to compare our resulting program with the existing imperative version — but we have ended up with a radical re-thinking of how to describe and evaluate contracts.

Though there is a great deal of work on domain-specific programming languages (see [9, 15] for surveys), our work is virtually the only attempt to give a formal description to financial contracts. An exception is the RISLA language developed at CWI [16], an object-oriented domain-specific language for financial contracts. RISLA is designed for an object-oriented framework, and appears to be more stateful and less declarative than our system.

We have presented our design as a combinator library embedded in Haskell, and indeed Haskell has proved an excellent host language for prototyping both the library design and various implementation choices. However, our design is absolutely not Haskell-specific. The big payoff comes from a declarative approach to *describing* contracts. As it happens we also used a functional language for *implementing* the contract language, but that is somewhat incidental. It could equally well be implemented as a free-standing domain-specific language, using domain-specific compiler technology. Indeed, one of us (Eber) has work afoot do to just this, compiling a contract into code that should be as fast or faster than the best available current valuation engines.

Although Haskell is lazy, and that was useful in our implementation, the really significant feature of the contract-description language is that it is *declarative* not that it is lazy. Our design is certainly not Haskell-specific.

There is much left to do. We need to expand the set of contract combinators to describe a wider range of contracts; to expand the set of observables; to provide semantics for these new combinators; to write down and prove a range of theorems about contracts; to consider whether the notion of a "normal form" makes sense for contracts; to build a robust implementation; to enable to apply easily the dramatic simplifications that closed formulas make possible; and to validate all this in real financial settings. We have only just begun.

Acknowledgements

We warmly thank John Wisbey, Jurgen Gaiser-Porter, and Malcolm Pymm at Lombard Risk Systems Ltd for their collaboration. They invested a great deal of time in educating two of the present authors (Peyton Jones and Seward) in the mysteries of financial contracts and the Black-Derman-Toy evaluation model. Jean-Marc Eber warmly thanks Philippe Artzner for many very helpful discussions. We also thank Conal Elliott, Andrew Kennedy, Stephen Javis, Andy Moran, Colin Runciman, and the ICFP referees, for their helpful feedback.

References

[1] P. Boyle, M. Broadie, and P. Glasserman. Monte carlo methods for security pricing. *Journal of Economic Dynamics and Control*, 21:1267–1321, 1997.

[2] K Claessen and D Sands. Observable sharing for functional circuit description. In PS Thiagarajan and R Yap, editors, *Advances in Computing Science (ASIAN'99); 5th Asian Computing Science Conference*, Lecture Notes in Computer Science, pages 62–73. Springer Verlag, 1999.

[3] B Cook and J Launchbury. Disposable memo functions. In J Launchbury, editor, *Haskell workshop*, Amsterdam, 1997.

[4] B Cook, J Launchbury, and J Matthews. Specifying superscalar microprocessors in hawk. In *Formal techniques for hardware and hardware-like systems*, Marstrand, Sweden, 1998.

[5] J. C. Cox, S. A. Ross, and M. Rubinstein. Option pricing: a simplified approach. *Journal of Financial Economics*, 7:229–263, 1979.

[6] C Elliott and P Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 263–273. ACM, Amsterdam, August 1997.

[7] S Finne, D Leijen, E Meijer, and SL Peyton Jones. Calling hell from heaven and heaven from hell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 114–125, Paris, September 1999. ACM.

[8] T. Ho and S. Lee. Term Structure Movements and Pricing Interest Rate Contingent Claims. *Journal of Finance*, 41:1011–1028, 1986.

[9] P Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, December 1996.

[10] John Hughes. Lazy memo-functions. In *Proc Aspenas workshop on implementation of functional languages*, February 1985.

[11] S Marlow, SL Peyton Jones, and C Elliott. Stretching the storage manager: weak pointers and stable names in haskell. In *International Workshop on Implementing Functional Languages (IFL'99)*, Lecture Notes in Computer Science, Lochem, The Netherlands, 1999. Springer Verlag.

[12] M. Musiela and M. Rutkowski. *Martingale Methods in Financial Modelling*. Springer, 1997.

[13] SL Peyton Jones, RJM Hughes, L Augustsson, D Barton, B Boutel, W Burton, J Fasel, K Hammond, R Hinze, P Hudak, T Johnsson, MP Jones, J Launchbury, E Meijer, J Peterson, A Reid, C Runciman, and PL Wadler. Report on the programming language Haskell 98. `http:/haskell.org`, February 1998.

[14] D. Revuz and M. Yor. *Continuous Martingales and Brownian Motion*. Springer, 1991.

[15] A van Deursen, P Kline, and J Visser. Domain-specific languages: an annotated bibliography. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam, 2000.

[16] A van Deursen and P Klint. Little languages: little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[17] P. Willmot, J.N. Dewyne, and S.D. Howison. *Option Pricing: Mathematical Models and Computation*. Oxford Financial Press, 1993.