Describing, manipulating and pricing financial contracts: The MLFi language

Centre for Financial Research, Judge Institute of Management
Cambridge, 14 March 2003

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

Agenda

- The problem
- Goals
- An algebra of contracts and observables (simplified)
- Reasoning about contracts
- Managing contracts
  - Fixings, exercise decisions
- Pricing contracts
  - Stochastic process algebra
  - Model capabilities, model implementations

The problem: Front-office vs back-office statements

What is a "Euribor 3M contract" ?

- Front-office, "Quant" answer:

  "This is a forward contract. By simple arguments, it can be shown that this depends only on the price of two ZC bonds $P(t, T_1)$ and $P(t, T_2)$.

  It is therefore an (adapted) stochastic process, function of these two former processes... The maths are beautiful... For the institutional details, ask the back-office guys..."

- Back-office answer:

  "This is a number that is officially published on each trading day at ... pm, and is available on ... . Many contracts are written on it and we must track these fixings... For the maths, ask the front-office guys..."

Some remarks about these two answers

- The question was: what is a ... contract ?
  - The "quant's" answer:

    how to calculate its price

    Contract = definition of the price process as a function of simpler elements (the ZC Bonds)
  - The back-office's answer:

    where to find its quote

    Contract = time series of daily quotes
- Nobody answered the question properly:

  give the "correct" definition of this forward contract!
- The MLFi technology aims at reconciliating these two diverging approaches...

Goals

- Allow for an exhaustive and sufficiently precise definition of (bilateral) financial contracts
  - Financial contract specification language
- Systematically derive pricing and operationnal management from the contract's definition
  - This derivation should itself be mathematically rigorous
  - In particular, contract definitions should be independent from the pricing mechanism
  - Implies the existence of a valuation model that does not depend on the contract's definition
- Reasoning about contracts. Is it possible, for instance, to simplify them ?

Part I: The MLFi specification language in 20 minutes

- Peculiarities of the finance domain
- Contract and observable combinators (simplified)
- A compositional contract algebra: contract definitions as values
- Algebraic contract transformations, “normal” form, sharing
- Temporal contract analysis, temporal simplifications
- From a few combinators to a whole language
- Market rules, industry-specific contract libraries

The importance of time

- Time plays a crucial role:
  - All financial contracts are strongly time-related
  - There is a natural total order on time

```
#   let t1 = 2002-12-20T16:00 (* ISO 8601 notation *) ;;
val t1 : date = 2002-12-20T16:00
#   let t2 = t1 ‘+days‘ 20.0 ;;
val t2 : date = 2003-01-09T16:00
#   t1 < t2 (* t2 is clearly ‘‘later’’ than t1 *);;
- : bool = true
```

- Time (values of type `date`) is a build-in feature of MLFi and largely exploited for checking time consistency of contract definitions (more about that later)

Important facts about contract definitions

- A contract is defined, but...
- ...the economic consequences for its holder depend on when the contract was acquired (acquisition date): think just about a bond bought years after its issue date
- All elementary definition blocks will therefore be parametrised by the acquisition date
- Most of the time, observables are “values” that are unknown when the contract is being defined and are observed (fixed) later by means of an agreed-upon procedure
- Contracts are written on such observables
- Observables may have different types (`float` for an index, `bool` for a default event)

- We present a minimalistic set of contract combinators:
  - In real life, there will be more of them, but...
  - ... the principles can be explained in a limited amount of time with a subset of them!

Precise contract combinators

```
val zero : contract
(** [zero] is a contract that, when acquired, carries no right or
   obligation. *)

val one : currency -> contract
(** Unitary currency payout. [one k] is a contract that, when
   acquired, pays immediately the holder one unit of the currency
   [k]. *)

val (‘and‘) : contract -> contract -> contract
(** Simultaneous immediate acquisition of two contracts.
   To acquire [c1 ‘and‘ c2] is the same as acquiring [c1] and
   [c2]. You acquire the rights and obligations of both contracts. *)
```

```
val either : (string * contract) list -> contract
(** Choice among many contracts.  To acquire [either [(s1,c1);
   (s2,c2);... (si,ci);... (sn,cn)]] means having the obligation to
   acquire exactly one of the [ci]s immediately. The necessarily
   different [si]s are used for managing the contract. They provide a
   unique identifier for each possible choice. *)

val (‘or‘) : contract -> contract -> contract
(** Binary short notation for the [either] operator. Management tags
   are set to [first_tag] and [second_tag] respectively. *)

val scale : float observable -> contract -> contract
(** Scaling a contract by an observable.
   If you acquire [scale o c], then you acquire [c] at the same
   moment, except that all of [c]’s payments are multiplied
   by the value of the observable [o] at the moment of acquisition. *)
```

```
val acquire : bool observable -> contract -> contract
(** Forced acquisition at entry in a region.
   [acquire r c] means that you must acquire [c] as soon as region
   [r] becomes [true].
   Note the particular case [acquire {[t]} c], where [t] is a date:
   Because [{[t]}] denotes the trivial region that is [true] at date [t] and
   [false] everywhere else, acquiring [acquire {[t]} c] means acquiring [c]
   at [t]. *)
```

We will only use the simplest form:

`acquire {[t]} c`,

meaning that you acquire contract `c` at date `t`.

Precise observable combinators

- Observables may be constant. “ ~ ” is a convenient notation for constant observables:

```
#   let cst_obs = 150.~ ;;
val cst_obs : float observable = ( 150.~)
```

- Functions of observables are again observables:

```
...
val ( +.~ ) : float observable -> float observable -> float observable
val ( -.~ ) : float observable -> float observable -> float observable
val ( *.~ ) : float observable -> float observable -> float observable
val ( /.~ ) : float observable -> float observable -> float observable
...
#   let another_obs = cst_obs +.~ 12.~ ;;
val another_obs : float observable = ( 162.~)
```

- Time is observable:

```
val time : date observable
(** Time observable.
   [time] is the [date] observable having, at each date {t}, value {t}. *)
```

- Observables may be annotated with a market (management) identifier

```
val market : string -> 'c observable -> 'c observable
(** Named Market Observable.
   Observables may be used both in contact management and in pricing.
   A label ([id]) is given to the observable to enable contract
   management. The [id] label is used in all contract management
   operations, for example, to record fixings.
   The label is ignored in {pricing}: [market id o] is the same
   as observable [o], which serves as the pricing model's underlying
   variable. *)
```

A contract algebra: contract definitions as values

We are used to manipulating, say, numbers and strings:

```
#   let r1 = 12 + 5
#   let r2 = "Happy" ^ " " ^ "in Cambridge"
#   let r3 = r1 + String.length r2 ;;
val r1 : int = 17
val r2 : string = "Happy in Cambridge"
val r3 : int = 35
```

But we can do the same kind of manipulation with contract definitions:

```
#   let c1 = one EUR
#   let c2 = one GBP
#   let c3 = c1 ‘and‘ c2 ;;
val c1 : contract = ((* horizon=max_date *) one EUR)
val c2 : contract = ((* horizon=max_date *) one GBP)
val c3 : contract = ((* horizon=max_date *) (one EUR) ‘and‘ (one GBP))
```

Simple contract definition

We want to specify the following contract:

One has the right, on 2003-04-22, to choose between receiving USD 100.000 on 2005-03-20, or receiving GBP 55.000 on 2005-06-30 (kind of European FX-option).

That's easy...

```
#   let usd_payment = acquire {[2005-03-20]} (scale 100000.~ (one USD)) ;;
val usd_payment : contract =
  ((* horizon=2005-03-20 *)
  acquire ({[2005-03-20]}) (scale 100000.~ (one USD)))

#   let gbp_payment = acquire {[2005-06-30]} (scale  55000.~ (one GBP)) ;;
val gbp_payment : contract =
  ((* horizon=2005-06-30 *)
  acquire ({[2005-06-30]}) (scale 55000.~ (one GBP)))

#   let choice = usd_payment ‘or‘ gbp_payment ;;
val choice : contract =
  ((* horizon=2005-03-20 *)
  (acquire ({[2005-03-20]}) (scale 100000.~ (one USD)))
  ‘or‘
  (acquire ({[2005-06-30]}) (scale 55000.~ (one GBP))))
```

```
#   let option = acquire {[2003-04-22]} choice ;;
val option : contract =
  ((* horizon=2003-04-22 *)
  acquire ({[2003-04-22]})
   ((acquire ({[2005-03-20]}) (scale 100000.~ (one USD)))
    `or`
    (acquire ({[2005-06-30]}) (scale 55000.~ (one GBP)))))
```

- From understanding only precisely what the basic combinators (`acquire`, `one`, `` `or` ``,...) mean, you derive the meaning of this contract
- That's the idea of algebraic expressions (like $(x + y)$)...
- ... and MLFi is doing the same for contracts !
- It's important to understand carefully how the `acquire` primitive is "organizing" the temporal decision and payment structure of our example contract

Temporal structure verification

The MLFi compiler checks the coherence of a contract’s temporal structure

Let’s return to our previous example and change it so that the option’s maturity falls after the underlying payment dates:

```
#   choice;;
- : contract =
((* horizon=2005-03-20 *)
(acquire ({[2005-03-20]}) (scale 100000.~ (one USD)))
‘or‘
(acquire ({[2005-06-30]}) (scale 55000.~ (one GBP))))
```

```
#   let option = acquire {[2006-04-22]} choice ;;
Exception:
Mlfi_contract.Acquire_incompatible_horizons (2006-04-22, 2005-03-20).
```

Algebraic contract simplifications

The MLFi compiler tries to simplify contract descriptions, and to share commonalities between contract sub-parts:

```
#   let c1 = (scale 1.~ (one EUR)) ‘and‘ zero ;;
val c1 : contract = ((* horizon=max_date *) one EUR)

#   let c2 = (scale 3.~ ((one EUR) ‘and‘ (one GBP))) ‘or‘
#      (scale 5.~ ((one GBP) ‘and‘ (one EUR))) ;;
val c2 : contract =
  ((* horizon=max_date *)
  let id0 = (one EUR) ‘and‘ (one GBP) in
  (scale 3.~ id0) ‘or‘ (scale 5.~ id0))
```

The system tries to represent contracts in a “normalized” way...

...but, for now, the MLFi compiler only performs simplifications that are compatible with the back-office

Temporal contract simplifications

Temporal structure simplifications are less trivial, as they analyse a contract definition along its potential temporal evolution

```
#   let c1' = gbp_payment ;;
val c1' : contract =
  ((* horizon=2005-06-30 *)
  acquire ({[2005-06-30]}) (scale 55000.~ (one GBP)))

#   let c2' = acquire {[2003-05-30]} c1' ;;
val c2' : contract =
  ((* horizon=2003-05-30 *)
  acquire ({[2003-05-30]})
   (acquire ({[2005-06-30]}) (scale 55000.~ (one GBP))))

#   normalize c2' ;;
- : contract =
((* horizon=2005-06-30 *) acquire ({[2005-06-30]}) (scale 55000.~ (one GBP)))
```

- The `normalize` function assumes implicitly that its contract argument is acquired at the earliest possible date (`min_date` in MLFi jargon), because it must make an initial assumption for temporal structure analysis

  Realistic contracts generally begin with an `acquire {[...]}` construct anyway, which precisely defines the beginning of the contract

- Important to remind: some simplifications apply to contracts, others to “acquired contracts”!

From contract combinators to a contract description language

We typically write functions that manipulate numbers:

```
#   let f x y z = x * (y + z) ;;
val f : int -> int -> int -> int = <fun>
```

And we can then use this function:

```
#   let r = f 2 5 3 ;;
val r : int = 16
```

We can do the same for contracts:

```
#   let scale_and o con1 con2 = scale o (con1 ‘and‘ con2) ;;
val scale_and : float observable -> contract -> contract -> contract = <fun>
```

And use this function:

```
#   let r = scale_and 150000.~ c1 c2 ;;
val r : contract =
  ((* horizon=max_date *)
  let id0 = (one EUR) ‘and‘ (one GBP) in
  scale 150000.~ ((one EUR) ‘and‘ ((scale 3.~ id0) ‘or‘ (scale 5.~ id0))))
```

There is nothing new here: all this machinery is just an easy way of easily combining contracts from simpler ones (or elementary ones)

Let’s write a function taking as input a list of (date, float) pairs and returning the contract paying all of these amounts (in, say, GBP):

```
#   let rec gbp_pays = function
#   | [] -> zero
#   | (t, amount) :: rest ->
#     (gbp_pays rest) ‘and‘
#     (acquire {[t]}
#        (scale (obs_of_float amount) (one GBP))) ;;
   val gbp_pays : (date * float) list -> contract = <fun>
```

and use it:

```
#   let r = gbp_pays
#   [(2001-01-15, 120.); (2002-01-14, 110.); (2003-01-16, 150.)] ;;
val r : contract =
  ((* horizon=2001-01-15 *)
  ((acquire ({[2003-01-16]}) (scale 150.~ (one GBP)))
   ‘and‘
   (acquire ({[2002-01-14]}) (scale 110.~ (one GBP))))
```

```
‘and‘
(acquire ({[2001-01-15]}) (scale 120.~ (one GBP))))
```

Product libraries, market rules

All this machinery is the basis for building realistic and complete libraries of schedule manipulations, market rules and contract definitions:

```
type is_business_day = date -> bool
(** Returns [true] if the argument is a business day, [false] otherwise...
```

…

```
instrument cashflow : amount * date -> contract
(** Acquire [amount] at date [date]. *)
```

…

```
val callput : callput -> contract -> amount -> period -> contract
(** [callput callputflag c strike (dbegin, dend)] is the
   right to buy ([callputflag = Call]) or sell ([callputflag = Put]) contract
   [c] against payment of [strike] in time interval
   ([dbegin], [dend])... *)
```

…

```
val bermudacallput : callput -> contract -> (date * amount) list -> contract
(** [bermudacallput callputflag c rule]:
   right to buy [callputflag=Call] or sell [callputflag=Put] contract [c] at
   dates and prices specified by schedule [rule] consisting of
   (date, strike) pairs with increasing dates. *)

...

val adj_raw_schedule :
    raw_schedule -> is_business_day -> business_day_convention -> raw_schedule
(** Returns an adjusted schedule from a given schedule by imposing the
   application of the business day and business day convention arguments. *)
```

Part II: Managing contracts

With a good understanding of our contract combinators, we want to manage a contract over time.

For exercise decisions for instance, we generalise the intuitive idea that a contract of the form

```
c1 ‘or‘ c2
```

should “reduce” to `c1` if:

- I acquired this contract, and
- I choose the “`c1` branch”

Fixings are treated similarly, by replacing the unknown observable identifier by its value in the contract’s definition

It is fundamental to understand that such a transition returns a modified contract that represents the new rights and obligations of the holder!

Contract management theory: operational semantics

$$\frac{}{\Phi \vdash < t, acquire\ t'\ c > \longrightarrow < t', \mathcal{S}[c] >}\ (SimplAcquire)$$

$$\frac{o \neq konst(x)}{\Phi \vdash < t, scale\ o\ c > \longrightarrow < t, \mathcal{S}[scale\ konst(obs_t(o))\ c] >}\ (ScaleFreeze)$$

$$\frac{\Phi \vdash < t, c > \xrightarrow{[.,ls,TR\ q\ k]} < t', c' >}{\Phi \vdash < t, scale\ konst(x)\ c > \xrightarrow{[.,ls,TR\ x*q\ k]} < t', \mathcal{S}[scale\ konst(x)\ c'] >}\ (ScaleQuant)$$

$$\frac{\Phi \vdash < t, c1 > \xrightarrow{e1} < t1, c1' > \qquad \Phi \vdash < t, c2 > \xrightarrow{e2} < t2, c2' > \qquad t1 \leq t2}{\Phi \vdash < t, c1\ and\ c2 > \xrightarrow{e1} < t1, \mathcal{S}[c1'\ and\ c2] >}\ (AndLeft)$$

$$\frac{}{\Phi \vdash < t, c1\ or\ c2 > \xrightarrow{[.,Long,XL]} < t, \mathcal{S}[c1] >}\ (OrLeft)$$

Manage the contract

- For simplicity, we assume the existence of the function:

```
val manage : contract -> manage_step -> contract * managed_step = <fun>
```

- `manage_step` can represent fixings, exercise decisions, time related events, barrier crossings
- Remember our FX option:

```
#   option ;;
- : contract =
((* horizon=2003-04-22 *)
acquire ({[2003-04-22]})
 ((acquire ({[2005-03-20]}) (scale 100000.~ (one USD)))
  ‘or‘
  (acquire ({[2005-06-30]}) (scale 55000.~ (one GBP)))))
```

- Now let's manage this contract. We first indicate an exercise decision (here the "first" choice available):

```
#   let after_choice, step =
#     manage option (Mgt_evt(2003-04-22, ("", Ev_exer "first"))) ;;
val after_choice : contract =
  ((* horizon=2005-03-20 *)
  aggregate (acquire ({[2005-03-20]}) (scale 100000.~ (one USD))))
val step : managed_step = Step_evt (2003-04-22, ("", Ev_exer "first"))
```

- Note that attempting to apply a wrong event (we are specifying an exercise date that is incompatible with the contract) results in an error:

```
#   let after_choice, step =
#     manage option (Mgt_evt(2003-04-23, ("", Ev_exer "first"))) ;;
Exception: Mlfi_manage.Irrelevant_event (2003-04-23, ("", Ev_exer "first")).
```

- We now move the clock forward so that any due payment can be processed:

```
#   let (after_pay, step) =
#     manage after_choice (Mgt_exec(2007-04-22)) ;;
val after_pay : contract = ((* horizon=max_date *) zero)
val step : managed_step =
  Step_exec (2007-04-22,
   [({t_acquired = 2005-03-20; t_q = 100000.; t_qe = ( 100000.~);
      t_aggregation_class = 1; t_treenode = ""; t_given = false},
    Transfer USD)])
```

- We may also ask the system to list pending events. Let's query our initial FX option:

```
#   let option_calendar = calendar option ;;
val option_calendar : calendar =
  {pendings =
    [({acquired = ( {[2003-04-22]}); abandon = ( false~); hyps = [];
       q = ( 1.~); treenode = ""; given = false},
     Pending_exer ["first"; "second"])];
   actions =
    [({acquired = ( {[2005-06-30]}); abandon = ( false~);
       hyps =
        [Hyp_exer ("second",
          {acquired = ( {[2003-04-22]}); abandon = ( false~); hyps = [];
           q = ( 1.~); treenode = ""; given = false})];
       q = ( 55000.~); treenode = ""; given = false},
     Transfer GBP);
     ({acquired = ( {[2005-03-20]}); abandon = ( false~);
       hyps =
        [Hyp_exer ("first",
          {acquired = ( {[2003-04-22]}); abandon = ( false~); hyps = [];
```

```
      q = ( 1.~); treenode = ""; given = false})];
  q = ( 100000.~); treenode = ""; given = false},
Transfer USD)]}
```

- The former result shows that the argument contract has an embedded option (`Pending_exer` in the `pendings` list), giving all necessary information (especially the option's maturity, here `2003-04-22`)

- Forthcoming payments are identified in the `actions` part and clearly identified as hypothetical (their existence depends on the previous exercise decision) because their hypotheses sets (`hyps`) are not empty and describe precisely the decision that must be taken for the payment to occur

- This feature provides a powerful way of checking properties for any contract described in MLFi. For instance:
    - A firm contract is simply defined as having no `Pending_exer` in his `pendings` list
    - Similarly, one can easily check that a given contract consists only of simple cash flows by checking that the `actions` part contains only `Transfers` with empty `hyps` sets

Part III: Pricing

- Pricer = f (contract, model)
- We define the notion of a model capabilities:
  - What currency(ies) and underlyings (observables) does the model support ?
  - What closed forms does the model support ? Closed form definitions are part of the model, not the contract!
  - Information about the model's geometry: PDE, Monte Carlo, etc.
- We generate a process expression, an abstract representation of the price (stochastic) process corresponding to the contract
  - Potential closed forms are resolved at this stage
  - More simplifications are applied on this expression
  - The process expression is then used to generate source code that will be linked with "low level" model primitives