Formal manipulation of financial contracts and evaluation models

University Finance Seminar
Cambridge, Judge Institue of Management Studies
21 January 2000

Jean-Marc Eber
Société Générale, Infi/Dir, Paris
jean-marc.eber@socgen.com

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

Implementation and industrial use is very slow, compared to "research"

Industrial users (trading-rooms) needs the methods applied to *"real" financial products* (micro precision). Institutional calculation rules may be complicated.

They want to measure effects on a whole *portfolio* (macro effect).

Spreadsheet approach is *no* longer acceptable for going "industrial" (back-office, risk measure, regulators,...).

Cost of entry for a new "industrial" financial product, model or method is huge.

We should (try to) lower this barrier...but still provide needed *flexibility*.

Finance technology today

"*Applied*" finance theory well understood, huge theoretical unification in progress.

More and more "common knowledge":

- its nice to play with stochastic processes, but...
- future competition will be on implementation, *not* mathematics !

Dissemination (repeated specificaton) of financial domain specific knowledge:

- each interest rate pricer reimplements the institutional rate calculation rules
- ...

Dissemination of implementations:

- rigourously identical numerical procedures implemented for different underyings (pde solver)
- ...

Absence of up to date documentation:

- what does it mean to be (a) correct (implementation) ?

No *formal specification tool* adapted to financial domain available.

Future of financial industry

Global approach for risk analysis and management. But what does "global" mean ?

Highly competitive, pressure on cost.

Regulatory pressure (documentation, correct implementation, operational risk factor,...).

Secured and standard, highly automated, deal description exchange format (even for OTC products).

- FpML is a tentative example

Migration of "trading-room" techniques to more "classical" banking activities:

- less "complicated", but less "standard" structures
- fewer closed-form solutions, more numerical approximations

Future of financial tools

Diversity of models, because diversity of needs, but unified deal (or position) description.

Incomplete market models or "methods":

- "Come-back" of classical *optimisation* methods, stochastic programming.

Evaluation of a financial contract *along "factor paths"* (back-testing, simulation for customers,...):

- Even if your pricing model is *not* path-dependant, this simulation *may be* !
- Marketing importance of such simulation approaches.

Merge of qualitative and quantitative analysis.

Technical *data-exchange format* standard (compare with other technical industries!)?

Computers more and more powerful (evidence, of course, but: *relative cost* of some algorithms or approaches are changing *enormously*).

Financial specification that adapts to and survives technological shifts:

- change of financial model
- change of computer architecture
- capacity to incorporate existing technology (components)

Description and evaluation

A financial contract is described by a *language*.

This (formal) language has a *syntax*.

This language can have one (or more) attached *semantic(s)* ("meaning")...

Such a description can be used in many different ways: it should be "exhaustive".

The semantics should be *compositional*.

Why a new language ?

Pragmatic reasons ("art"):

- good engineering practice...
- well adapted to application domain
- Conciseness (example: object oriented programming)

Fundamental reasons ("science"):

- simplify semantics, implementation, orthogonality
- reduce the complexity class of the language

Properties of compositional descriptions of contracts

Enables to *distinguish* between (legal, say) *description* and *evaluation* (of such a description).

Predominance of a particular "object": time

- "solution algorithms" go monotonically along the time line, but also...
- from irreversibility of time comes the typical *recursive* contract *description* "à la" *dynamic programming*.

Its impossible to write "impossible" contracts (like a "looping" program in a classical programming language).

Verified to be "bug free" by, for instance, *lawyers*!

- make the contract *enforceable*: only a *finite* number of actions or events.

Real world financial contracts

We should keep previous advantages when extending the language to real world financial contracts.

What we certainly have to do:

- link our language with the "outside" world, especially for writing contracts on *observables*.
- introduce *schedules*, and a way to use them concisely and efficently.
- mention *models* and model specialisation (closed form solutions for instance).

Observables

Financial contracts pay-outs are always written on *observables*:

- an equity quoted spot, a quoted future contract, an interest rate like 3mLibor, temperature (in degree celsius) at Notre Dame in Paris, rating class of a given corporate, default of a corporate on his debt,...

Observables are defined as being common information and easily verified:

- Necessary condition for being legally enforceable

Observable aren't necessarly real valued (see the last two examples: enumerated type, boolean).

Applying an arithmetic function to observables generates another observable.

Observables don't have a currency: the *contract* on an observable prescribes a payment in a given currency!

- `c = (quote T GBP obs)` is a contract that pays immediately obs *pounds*.

- quanto structures, for instance, can only be written on observables:
  `quanto = (quote T GBP socgen)`

Constants and elapsed time are observables.

Lagged observables are observables (path dependant products).

Uniquely identified observables are the (pricing) link between front- and back-office.

Observables and pricing

Observables often play the role of "underlyings", of course.

A model must *implement* all the observables referenced by a contract for pricing it.

But aren't some observables simply a function of other *prices* ?

- think about a forward rate being "equal" to a function of two discount bonds:
- no! an observable is always an entity on itself. but...
- ...a model (or a class of models) may, for pricing purposes *only*, declare such a (no-arbitrage) relationship.

A model *implements* an observable by:

- implementing it directly, as a function of its state variables (the "S" of Black-Scholes)
- declaring it a function of other observables
- declaring it a function of other prices

Schedules

We take the simplest example: a sum of discount bonds in the same currency:

(zcb date "12jan2002" 10.0 GBP) 'and' (zcb date "11jan2003" 11.0 GBP) 'and' (zcb date "9jan2004" 12.0 GBP) 'and' (zcb date "12jan2005" 8.0 GBP)

We want to create a combinator that takes as input a schedule of (date, float) pairs, and generates the sum:

schedule = [
(date "12jan2002", 10.0),
(date "11jan2003", 11.0),
(date "9jan2004", 12.0),
(date "12jan2005", 8.0)
]

`bond_GBP = newoperator schedule`

Schedules and iteration

Schedules are lists. Well known data type: is actively studied by computer scientists as the simplest *recursive datastructure*.

A huge literature exists about efficent manipulation of (or "calculating" with) lists.

In finance, schedules are ordered along time.

A contract definition is obtained by a *monotone* iteration over a schedule.

`foldr` gives us what we need:
(('a list)×(('a×'b)→'b)×('a→'b))→'b
$foldr([x_1, x_2, ..., x_{n-1}, x_n], f, g) =$
$f(x_1, f(x_2, ..., f(x_{n-1}, g(x_n))...))$

If lists are only finite, foldr always *terminates*!

A few other primitives are needed, especially for building (finite) lists and transforming (finite) lists.

Schedule example

We want to define a bond_GBP, given a list `l` of pairs `(d, c)`, representing `c` pounds to be paid at date `d`:

```
g :: (Date * Float) -> Contract
g (t, a) = zcb t a GBP
```

```
f :: (Date * Float) * Contract -> Contract
f (t, a) c = (zcb t a GBP) 'and' c
```

then

```
bond_GBP = foldr l f g
```

This example is, of course, trivial: you can imagine much more complex structures (full compositional approach).

Important: the financial understanding of the foldr operator is a mechanic consequence of its definition.

foldr: explicit (structural) recursion operator for financial contract description

Explicit documentation (generate schedule diagrams, time charts...).

Expose regularity along the datastructure to a compiler: many optimisations possible.

But: highly abstract specification; reuse possibility, libraries of code.

Deforestation: gluing together many "passes" for describing an algorithm, but guaranty of efficiency!

Good starting point for automatic inductive proofs!

Semantics: what does a contract mean ?

For manipulating contracts (pricing is only one of this manipulations), we need a precise *understanding* of contracts, but...

- ...no (semantic) theory of contracts known to us...
- ...we have to build one!

An *axiomatic* theory of domination, denoted a $\geq$b, for contract a *dominates* contract b, and *equality* defined as mutual domination.

Defined inductively along the combinators; example:

- (c1 $\geq$ c2) $\wedge$ (d1 $\geq$ d2) $\Rightarrow$
  (c1 'and' d1) $\geq$ (c2 'and' d2)

Set of axioms driven by application: two contracts may be equal for one application, but not for another.

We call such a set of axioms a *theory*.

Typical applications "add" axioms, or *refine* the theory (example: always positive interest rates)!

Enables automatic transformation of contracts, because we have the following "replace equal by equal" possibility:

Theorem 0.1 *Contextual equality: if c1 = c2, then for any context C[], we have: C[c1] = C[c2].*

One possible theory: c1 $\geq$ c2 iff (price process of c1) $\geq$ (price process of c2) for any "reasonable" (no arbitrage) model ("Mertons Rational Option Pricing").

Abstract semantics: an application

Can we analyse the essence of American optionality *without* a model in mind ?

American option in terms of contracts only: "the sooner you get it, the better".

When is a contract c *early preferred* ? Our language has all the tools to express this:

Definition 0.1 *c is early preferred, written Early(c), iff:*
$\forall$`t`, `(truncate t c)` $\geq$ `(get(truncate t c))`

Then we define Late(c), TimeIndiff(c).

In a pricing semantics, this corresponds to the notion of *super-*, *sub-* and *martingales* of normalised price processes.

We postulate: $Early$(`anytime c`) and rules like:

- H(c)=H(`d`), $Early$(`c`) $\wedge$ $Early$(`d`) $\Rightarrow$ $Early$(`c 'and' d`)

Enables to deduce qualitative properties of contracts, and, thus, of associated (normalised) price processes.

Applies to less trivial contracts:

Exercice: define a Bermudan option structure on an underlying as a function `berm` of a schedule `rule` and a underlying contract `c`. Schedule `rule` is a list of triples (`begdate, enddate, strike`) defining the exercise period and strike for this period. Define `berm` with a foldr. Convince yourself that an inductive proof shows that *any* Bermudan option is early preferred!

Positive interest rates

Knowing that interest rates are always positive may be important.

This property isn't always true (just think about a linear gaussian interest rate model!).

Can we *abstractly* define positiveness of interest rates ?

Yes, it suffices to express the "preference for the present":

Definition 0.2 *We have positive interest rates for unit (or numeraire)* `k` *up to date* `T` *iff*
Early(`quote T k (const 1.0)`*)*

Abstract semantics: a classical proof revisited

Can we *prove* the "classical" result: European call on S = American call on S if S is a non dividend paying stock *and* interest rates are positive ?

Proof:

TimeIndiff(S) (assumption)

TimeIndiff(truncate T S) (th)

Early(quote T k (const K)) (positive interest rates)

Late(give quote T k (const K)) (axiom)

Late((truncate T S) 'and' give quote T k (const K) (axiom)

Late(truncate T (S 'and give quote T k (konst K))) (eq)

Late(truncate T (S 'and give quote T k (konst K)) or zero (...)) (axiom)

anytime(truncate T (S 'and give quote T k (konst K)) or zero (...)) = get (...)

Use of abstract semantics for pricing

Simplify numerical procedures:

- replace "American" style options on Early preferred contracts by "European" structures
- Use linearity etc... to optimise generated numerical code

Generate "good" time discretisation by applying a *discretisation strategy* to a contract definition.

Choice of numeraire problem: what is the "best" choice (can, and should, be done at the level of the abstract semantics) ?

Closed form solutions and evaluation

Closed form solutions (CF) are fundamental in finance: huge advantage, not only for pricing, but also for hedging.

Definition: easy to calculate as a "function" of state variables.

Algebraic combination of CFs is a CF: we should keep "hedging advantage".

CFs may be used *inside* a numerical procedure.

Being a CF *depends* on the model: the contract writer don't even know about that.

A model may "rewrite" a contract as a function of its state variables. It should keep this "representation" as high as possible in the contract "tree".

Formal differentiation should be used.

This approach is *necessary* if we want the best of both worlds (complete semantic description *and* efficency).

Contracts, semantics, evaluation in industrial practice

We work on the kernel language: it should be extended or be extendable (legal annotations, pricing annotations, embedd technical documentation ...).

Contract execution is (also) a document processing business.

An explicit contract description is the only method for a global and unified approach.

Enables flexibility, by implementing language processors for needs we even *don't know* today.

Describe the dynamic possible evolution of the contract in time (exercise decisions, rate fixings,...).

Models must also be described in a language.

Code generation tehnology.