Less power ⇒ more possibilities

Languages for high-performance computing

Peter Marks, Tim Williams – EDG Quantitative Analytics, Barclays

BARCLAYS

Derivative contracts

- An agreement for the issuer to pay money to the buyer, determined by the performance of one or more underlyings
- Vanillas: Call, Put, …
- Exotics: Autocall, Worst of Reverse Convertible, …
- Strategies: CPPI, Combined Revolver, …
- Formal specification of the payoff
- Pricing: Monte Carlo, PDE
- Lifecycling: payments, events, barrier monitoring, effective maturity,…

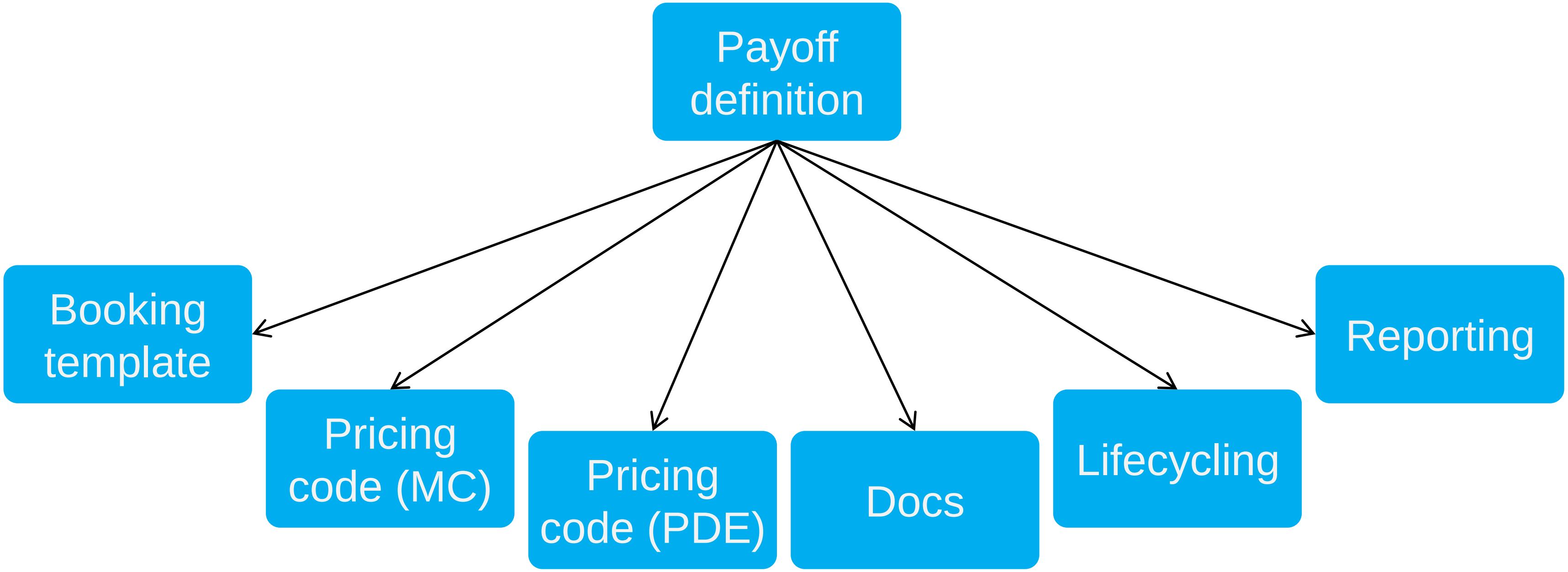
Traditional approach

Model (Excel)	Booking template	Pricing code (C)	Docs	Lifecycling	Reporting
Model (Excel)	Booking template	Pricing code (C)	Docs	Lifecycling	Reporting
Model (Excel)	Booking template	Pricing code (C)	Docs	Lifecycling	Reporting
Model (Excel)	Booking template	Pricing code (C)	Docs	Lifecycling	Reporting
Model (Excel)	Booking template	Pricing code (C)	Docs	Lifecycling	Reporting
Model (Excel)	Booking template	Pricing code (C)	Docs	Lifecycling	Reporting

Key issues

- Multiple payoff representations that must be consistent
- Very hard to check and ensure consistency
- New payoffs require changes in many systems
- Code is bespoke for each payoff and can be complex
- On-boarding a payoff requires the coordination of many teams and takes a long time

The FPF approach



Single payoff representation, generic tooling

A simple payoff – Call

On the *outDate*, pay the performance of the *asset* minus the *strike*, floored at 0, settled at the *payDate*

The performance is the price of the *asset* on the *outDate* divided by the price of the *asset* on the *inDate*

```
function call{asset, strike, inDate, outDate, payDate}
  on inDate
    inObs = asset
  end

  on outDate
    pay max(0, asset / inObs - strike) at payDate
  end
end
```

A vol controlled strategy

```
function volControl{asset, exposureCap, targetVol, triggerLvl, winSize, fee, dcfBasis, sched}
  calcSched = drop(winSize, sched)

  on calcSched
    rv = rollingUnbiasedRealizedVol(winSize, asset, sched)
    idealExposure = min(exposureCap, targetVol / rv)
  end

  on first(calcSched)
    index = 1
    targetExposure = idealExposure
  end

  on subsequent(calcSched)
    rebalance = abs(idealExposure - targetExposure) > triggerLvl
    targetExposure = if rebalance then idealExposure else targetExposure
    actualExposure = prev(targetExposure, calcSched)
    feeDCF = dcf(dcfBasis, -1, Accrual, sched)
    index = index * (1 + actualExposure * (cliquet(asset, schedule) - 1) - fee * feeDCF)
  end

  return index
end
```

An option on a vol controlled strategy

```
function volControlOption(params)
  index = volControl(params)
  return call{override asset = index | params}
end
```

Language features

- Very powerful type system with full inference
- Records, variants and statically sized arrays
- Syntactic sugar such as record field pattern matching
- Execution sequence aligned to real-world time
- Schedule guards and simple schedule manipulation operations
- Extensive looping capabilities including sorting and joining
- Functions define local state and can take independent schedules providing encapsulation leading to simple, safe, efficient composition

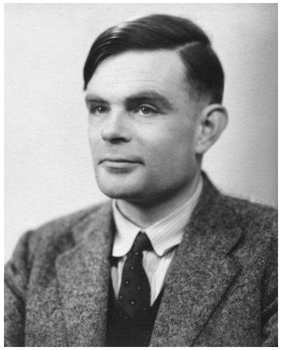
Language restrictions

- No dynamic memory allocation
- All arrays statically sized and regular
- No recursion or unbounded loops
- No higher-order functions
- No pointers or references
- No array index operator
- No arbitrary IO
- No FFI
- Variables must evolve forward and payments and exits are primitive
- Access to only current step market data
- All schedules statically known
- All underlyings statically known

Machines and computability

- A language is **Turing-equivalent**, if it can be used to model a Turing machine
- A computation is **Turing-complete**, if it can only be computed by a Turing-equivalent machine

How common are Turing-complete problems?



Computable functions

We only need to look at numeric computations (no loss of generality).

- **First-order primitive recursive functions** (computable by loop programs)

  For example: addition, division, factorial, exponential, fibonacci and most total functions that you might imagine

- **Higher-order primitive recursive functions** (computable by functional folds)

  For example: the Ackermann function is one of the simplest examples of a well-defined total function, that is not first-order primitive recursive

- **μ-recursive functions** (computable by a Turing machine)

  Adds a partial search operator μ, equivalent to general recursion

Abstract machines

Classify (a selection) by the formal language they can recognise:

Languages	Automaton
Recursively enumerable	Turing machine
Context-sensitive	Linear-bounded non-deterministic automaton
Indexed	Nested Stack automaton
Context-free	Non-deterministic pushdown automaton
Deterministic context-free	Deterministic pushdown automaton
Regular	Finite state automaton

The cost of power

Turing machines $\Leftrightarrow$ μ-recursive functions $\Leftrightarrow$ Untyped Lambda Calculus

Untyped Lambda expressions:

- cannot be compared for equivalence
- have no normal form
- cannot be analysed for termination (halting problem)

$$\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

BARCLAYS

Partial programming

By Gödel's incompleteness theorem, any language with enough power to host itself, cannot be consistent (false is derivable, through non-termination and $\perp$ in all types).

```
loop : Int → Int
loop n = 1 + loop n
```

$\Rightarrow$ `loop 0 = 1 + loop 0`

$\Rightarrow$ `0 = 1`

Restriction of power

Example : Simply-Typed Lambda Calculus

- Using types, disallow self-application $\lambda x.xx$ preventing the formation of fixed-point combinators
- All expressions strongly normalising, every reduction sequence terminates in a normal form
- Programs now isomorphic to logic proofs (Curry-Howard)

The power tradeoff

There is a dichotomy in language design, because of the halting problem. For our programming discipline we are forced to choose between

A) Security - a language in which all programs are known to terminate.

B) Universality - a language in which we can write
- (i) all terminating programs
- (ii) silly programs which fail to terminate and, given an arbitrary program we cannot in general say if it is (i) or (ii).

Five decades ago, at the beginning of electronic computing, we chose (B).

David Turner – Total Functional Programming, 2004

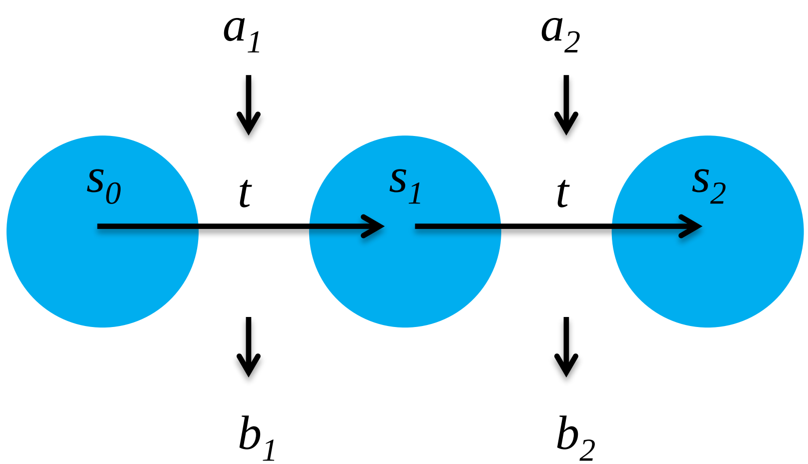
Restricted languages

Language	Restriction	Benefit
SQL	Abstracts the details of how data is stored and retrieved	Highly optimisable, portable, finite, access dynamically determined
RegEx	Uses a Regular Language	Can run in time O(*n*) on a string of size *n* (after optimisation)
HTML	(Originally) intended as semantic markup	Customisable presentation, accessibility
Functional Programming	No mutation or other side-effects	Compositionality, referential transparency
Theorem prover	Total functions	Propositions-as-types, Programs-as-proofs (Curry-Howard)

Lucid computation model

Idea : model a time-dependent price calculation using

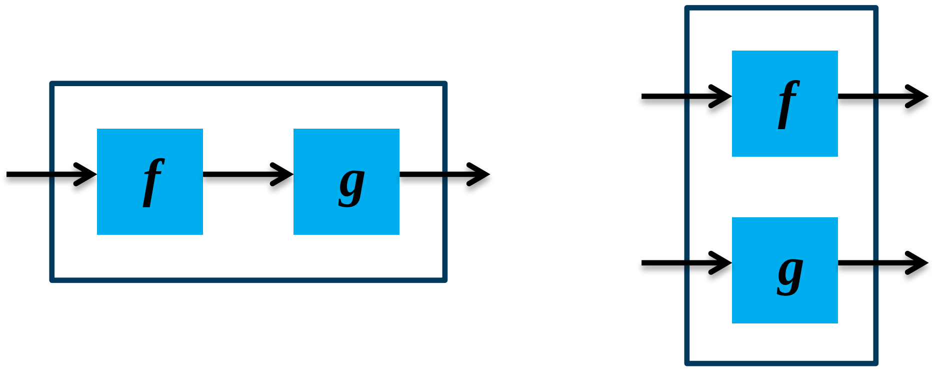
- an input type a (time-dependent data)
- an output type b (price metrics)
- a state type s (the calculations internal state)
- a transition function $t : (a, s) \rightarrow (b, s)$
- a start state s_0



Mealy Machines

Mealy machine – a finite-state machine whose output value is determined both by its current state and the current input.

- We understand the (always bounded) space and time of a single Mealy Machine
- We understand how Mealy Machines compose (in serial and in parallel)



The computation model of a Lucid program is a Mealy Machine

The benefits of restriction in Lucid

Restriction	Advantage	Benefit
No dynamic memory allocation	Statically known memory requirement	No OOM errors and farm memory optimisation
All arrays statically sized and regular	Static checking of array size matches	No runtime array size mismatch failures
No recursion or unbounded loops	Guaranteed termination	No hanging pricing
	Can be statically analysed	Many optimisations possible including beta reduction, eliminating the need for a stack
No higher-order functions	Don't need capture	Simple runtime system
No pointers or references	Encapsulation	Localised analysis and simple composition
	Safety	No Null Pointer Exceptions
	Abstraction	Model and platform independence

BARCLAYS

The benefits of restriction in Lucid (cont.)

Restriction	Advantage	Benefit
No array index operator	Array bounds safety	No Out of Bounds errors
No arbitrary IO	Independence	No IO failures
	Isolation	Allows more static analysis
No FFI	Closed system	Allows more static analysis and portability
Variables must evolve forward and payments and exits are primitive	Abstracted from execution	Can be priced in Monte Carlo or PDE, or used for accrual
Access to only current step market data	Enables time-slice Monte Carlo	More optimal for CPU, perfect fit for GPU
All schedules statically known	Can statically check dynamic data flow	No uninitialised data access
All underlyings statically known	Market data requirements statically known	Simplifies distribution

The payoff

- >20 backends in production use
  - MC pricing, PDE pricing, discontinuity analysis, payment reporting, TeX generation…
- 100s of combinations of flags
- >300 trade scripts with live positions
- Median time to market for new trade type <24 hours
- 580ms PDE pricing time for typical 3y daily-observed autocallable
  - Credit to FiDEs team
- New PTX generation backend developed in <2 weeks
- Monte Carlo pricing on GPU >200x faster than CPU
  - Credit to Supernova team

References

Turner, D. A. (2004). Total Functional Programming.

Meyer, A. R., & Ritchie, D. M. (1967). The complexity of loop programs.