Composing contracts
An adventure in financial engineering

Simon Peyton Jones and Julian Seward,
Microsoft Research
and
Jean Marc Eber, LexiFi Technologies
The big picture

Financial engineering

Programming language design and implementation

Jean Marc

Simon and Julian
The big picture

Swaps, caps, options, european, bermudan, straddle, floors, swaptions, swallows, spreads, futures

Financial engineering

Programming language design and implementation

Jean Marc

Simon and Julian
What we want to do

Precise description of a contract

- Scheduling for back office
- Valuation and hedging
- Legal and documentation
- etc....
What we want to do

Precise description of a pudding

- Compute sugar content
- Estimate time to make
- Instructions to make it

etc....
What we want to do

Precise description of a pudding

Compute sugar content

Estimate time to make

Instructions to make it

Bad approach

- List all puddings (trifle, lemon upside down pudding, Dutch apple cake, Christmas pudding)
- For each pudding, write down sugar content, time to make, instructions etc
What we want to do

Precise description of a pudding

Compute sugar content

Estimate time to make

Instructions to make it

Good approach

- Define a small set of “pudding combinators”
- Define all puddings in terms of these combinators
- Calculate sugar content from these combinators too
Creamy fruit salad

Whipped

On top of

Mixture

Chopped

Take

1 pint

Cream

Take

3

Apples

Optional

Take

6

Oranges

Combinators combine small puddings into bigger puddings
Trees can be written as text

Notation:
- parent child1 child2
- function arg1 arg2

```
salad = onTopOf topping main_part
topping = whipped (take pint cream)
main_part = mixture apple_part orange_part
apple_part = chopped (take 3 apple)
orange_part = optional (take 6 oranges)
```

Slogan: a domain-specific language for describing puddings
Building a simple contract

“Receive £100 on 1 Jan 2010”

\[
\text{c1 :: Contract} \\
\text{c1 = zcb (date “1 Jan 2010”) 100 Pounds}
\]

\[
\text{zcb :: Date -> Float -> Currency -> Contract} \\
\text{-- Zero coupon bond}
\]

Combinators will appear in blue boxes
Combing contracts

\[
c1, c2, c3 :: \text{Contract}
c1 = \text{zcb (date "1 Jan 2010") 100 Pounds}
c2 = \text{zcb (date "1 Jan 2011") 100 Pounds}
\]

\[
c3 = \text{and } c1 \text{ and } c2
\]

\[
\text{and :: Contract } \rightarrow \text{ Contract } \rightarrow \text{ Contract}
\quad \text{-- Both } c1 \text{ and } c2
\]
Inverting a contract

c4 = c1 `and` give c2

give :: Contract -> Contract
-- Invert role of parties

- `and` is like addition
- `give` is like negation
New combinators from old

\[
\text{andGive} :: \text{Contract} \rightarrow \text{Contract} \rightarrow \text{Contract} \\
\text{andGive } u1 \ u2 = u1 \ `\text{and}` \ \text{give} \ u2
\]

- \text{andGive} is a new combinator, defined in terms of simpler combinators.
- To the “user”, \text{andGive} is no different to a primitive, built-in combinator.
- This is the key to extensibility: users can write their own libraries of combinators to extend the built-in ones.
Indeed, **zcb** is not primitive:

```
defining zcb

zcb :: Date -> Float -> Currency -> Contract
zcb t f k = at t (scaleK f (one k))
```

---

**one** :: Currency -> Contract  
-- Receive one unit of currency **immediately**

**at** :: Date -> Contract -> Contract  
-- Acquire the contract **at specified date**

**scaleK** :: Float -> Contract -> Contract  
-- Scale contract by specified factor
Acquisition dates

one :: Currency -> Contract
-- Receive one unit of currency immediately

at :: Date -> Contract -> Contract
-- Acquire the underlying contract at specified date

- If you acquire the contract \((\text{one} \ k)\), you receive one unit of currency \(k\) immediately.
- If you acquire the contract \((\text{at} \ t \ u)\) at time \(s < t\), then you acquire the contract \(u\) at the (later) time \(t\).
- If you acquire \((\text{at} \ t \ u)\) later than \(t\), you get nothing.
Pay me $1000 \times (\text{the number of inches of snow} - 10) \text{ on 1 Jan 2002}

c :: Contract
\[ c = \text{at "1 Jan 2002" (scale scale\_factor (one Dollar))} \]

scale\_factor :: \text{Obs Float}
\[ \text{scale\_factor} = 1000 \times (\text{snow} - 10) \]

scale :: \text{Obs Float} \to \text{Contract} \to \text{Contract}
\quad \text{-- Scale the contract by the value of the observable}
\quad \text{-- at the moment of acquisition}
An **observable** is an objectively-measurable, but perhaps time-varying quantity, or a value derived from such measurements.

```hs
snow :: Obs Float
date :: Obs Date

const :: a -> Obs a
(*), (-) :: Obs Float -> Obs Float -> Obs Float
(>), (>=) :: Obs a -> Obs a -> Obs Bool

scaleK k c = scale (const k) c
```
Acquisition triggers

Acquisition can be triggered by a boolean observable

```
c :: Contract
  c = when late_snow (one GBP)

late_snow :: Obs Bool
  late_snow = date > const "1 Apr 2003" && snow > 100
```

```
when :: Obs Bool -> Contract -> Contract
  -- If you acquire (when o c), you acquire c at the
  -- first moment when o subsequently becomes True

at t c  = when (date == const t) c
```
An **option** gives the flexibility to

- **Choose which** contract to acquire (or, as a special case, **whether** to acquire a contract)

- **Choose when** to acquire a contract  
  (exercising the option = acquiring the underlying contract)
European option: at a particular date you may choose to acquire an “underlying” contract, or to decline

\[
european :: \text{Date} \to \text{Contract} \to \text{Contract}
\]
\[
european \ t \ u = \text{at} \ t \ (u \ `\text{or}` \ \text{zero})
\]

\[
\text{or} :: \text{Contract} \to \text{Contract} \to \text{Contract}
\]
\[
\text{-- Acquire your choice of either c1 or c2 immediately}
\]

\[
\text{zero} :: \text{Contract}
\]
\[
\text{-- A worthless contract}
\]
Reminder...

Remember that the underlying contract is arbitrary

c5 :: Contract
c5 = european t1 (european t2 c1)

This is already beyond what current systems can handle
The option to acquire 10 Microsoft shares, for $100, anytime between t1 and t2 years from now

```
anytime :: Obs Bool -> Contract -> Contract
-- Acquire the underlying contract at
-- any time the observable is True

golden_handcuff = anytime (date >= t1 && date <= t2) shares

shares = zero `or` (scaleK -100 (one Dollar) `and`
  scaleK 10 (one MSShare))
```
Summary so far

- Only 10 combinators (after many, many design iterations)
- Each combinator does one thing
- Can be combined to describe a rich variety of contracts
- Surprisingly elegant
But what does it all mean?

- We need an absolutely precise specification of what the combinators mean: their semantics.
- And we would like to do something useful with our (now precisely described) contracts.
- One very useful thing is to compute a contract’s value.
Use denotational semantics

- The **denotation** of a program is a mathematical value that embodies what the program “means”
- Two programs are **equivalent** if they have the same denotation
- A denotational semantics should be **compositional**: the denotation of \((P_1 + P_2)\) is gotten by combining somehow the denotations of \(P_1\) and \(P_2\)
Wanted: $S(P)$, the sugar content of pudding $P$

- $S(\text{onTopOf } p1 \ p2) = S(p1) + S(p2)$
- $S(\text{whipped } p) = S(p)$
- $S(\text{take } q \ i) = q \times S(i)$

...etc...

- When we define a new recipe, we can calculate its sugar content with no further work.
- Only if we add new combinators or new ingredients do we need to enhance $S$. 
Processing puddings

- Wanted: $S(P)$, the sugar content of pudding $P$

\[
\begin{align*}
S(\text{onTopOf } p1 \ p2) &= S(p1) + S(p2) \\
S(\text{whipped } p) &= S(p) \\
S(\text{take } q \ i) &= q \times S(i)
\end{align*}
\]

...etc...

$S$ is \textit{compositional}

To compute $S$ for a compound pudding,

- Compute $S$ for the sub-puddings
- Combine the results in some combinator-dependent way
What is the denotation of a contract?

Main idea: the denotation of a contract is a random process that models the value of acquiring the contract at that moment.

\[ \mathcal{E} : \text{Contract} \rightarrow \text{RandomProcess} \]

RandomProcess = Time -> RandomVariable

Uncertainty increases with time.
Compositional valuation

\[ \epsilon(c_1 \text{ `and` } c_2) = \epsilon(c_1) + \epsilon(c_2) \]
\[ \epsilon(c_1 \text{ `or` } c_2) = \max(\epsilon(c_1), \epsilon(c_2)) \]
\[ \epsilon(\text{give } c) = -\epsilon(c) \]
\[ \epsilon(\text{when } o \text{ c}) = \text{discount}(\epsilon(o), \epsilon(c)) \]
\[ \epsilon(\text{anytime } o \text{ c}) = \text{snell}(\epsilon(o), \epsilon(c)) \]
...

This is a **major payoff**! Deal with the 10-ish combinators, and we are done with valuation!
Reasoning about equivalence

- Using this semantics we can prove (for example) that
  \[ \text{anytime} \circ (\text{anytime} \circ c) = \text{anytime} \circ c \]

- Depends on algebra of random processes (snell, discount, etc).
  Bring on the mathematicians!
A compiler for contracts

1. Contract
2. Take semantics
3. Random process
4. Code generation
5. Valuation program
6. Transform using algebraic laws
Valuation

- There are many numerical methods to compute discrete approximations to random processes (*tons and tons and tons and tons and tons and tons and tons and tons and tons and tons* of existing work)

- Model of world (e.g. interest rates, snow fall)

- Valuation engine

- Contract
One possible evaluation model: BDT

zcbb 3 100 Pounds
contract C

interest rate model M

Valuation engine

Value tree $\mathcal{E}(C)$
Space and time

- Obvious implementation computes the value tree for each sub-contract
- But these value trees can get BIG
- And often, parts of them are not needed

simple discounting at t
Haskell to the rescue

“Lazy evaluation” means that

- data structures are computed incrementally, as they are needed (so the trees never exist in memory all at once)
- parts that are never needed are never computed

Slogan
We think of the tree as a first class value “all at once” but it is only materialised “piecemeal”
An operational semantics

- As time goes on, a contract evolves
e.g. \( zcb \ t1 \ n \ k \ `\text{and}` \ zcb \ t2 \ n \ k \)
- Want to value your current contract “book”
- So we want to say formally how a contract, or group of contract evolves with time, and how it interacts with its environment (e.g. emit cash, make choice)
- Work on the operational semantics of programming languages is directly relevant (e.g. bisimulation)
A compiler for contracts

- Contract
  - Take semantics
    - Random process
      - Code generation
        - Valuation program

Evolve using operational semantics
Section 1. The attorney shall provide, on a non-exclusive basis, legal services up to (n) hours per month, and furthermore provide services in excess of (n) hours upon agreement.

Section 2. In consideration hereof, the company shall pay a MDCC 16 monthly fee of (amount in dollars) before the 8th day of the following month and (rate) per hour for any services in excess of (n) hours 40 days after the receipt of an invoice.

Section 3. This contract is valid 1/1-12/31, 2008.
Again: a domain specific language

```haskell
letrec
extra (att, com, invoice, pay) =
    ( Success
      + transmit (att, com, invoice, T2).
      transmit (com, att, pay, T3 | T3 <= T2 + 45d))

legal (att, com, fee, invoice, pay, n, m, end) =
    transmit (att, com, H, T | n < T and T <= m).
    ( extra (att, com, invoice, pay)
      || transmit (com att, fee, T | T <= m + 8d)
      || ( legal (att, com, fee, invoice, pay, m, min(m + 30d,end), end)
          + transmit (att, com, end, T | end <= T)))
in
legal ("Attorney","Company",10000.invoice,pay,0,30,360)
```
Summary

- A small set of built-in combinators: named and tamed
- A user-extensible library defines the zoo of contracts
- Compositional denotational semantics, leads directly to modular valuation algorithms
- Risk Magazine Software Product of the Year Prize
- Jean-Marc has started a company, LexiFi, to commercialise the ideas. Paying customers, typesafe .NET interoperation, sophisticated pricing models etc.
Summary

- A small set of built-in combinators: named and tamed
- A user-extensible library defines the zoo of contracts
- Compositional denotational semantics, leads directly to modular valuation algorithms
- Risk Magazine Software Product of the Year Prize
- Jean-Marc has started a company, LexiFi, to commercialise the ideas

**Beats higher order logic hands down for party conversation**