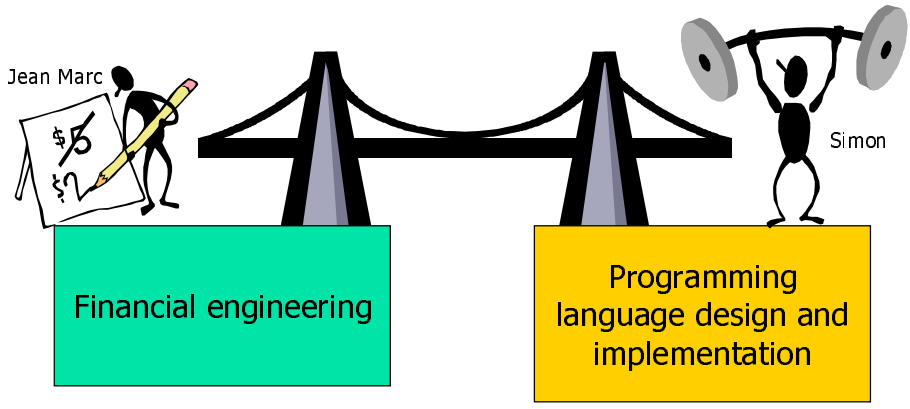
Compositional description and valuation of financial contracts

Simon Peyton Jones, Microsoft Research
and
Jean Marc Eber, Société Général

The big picture

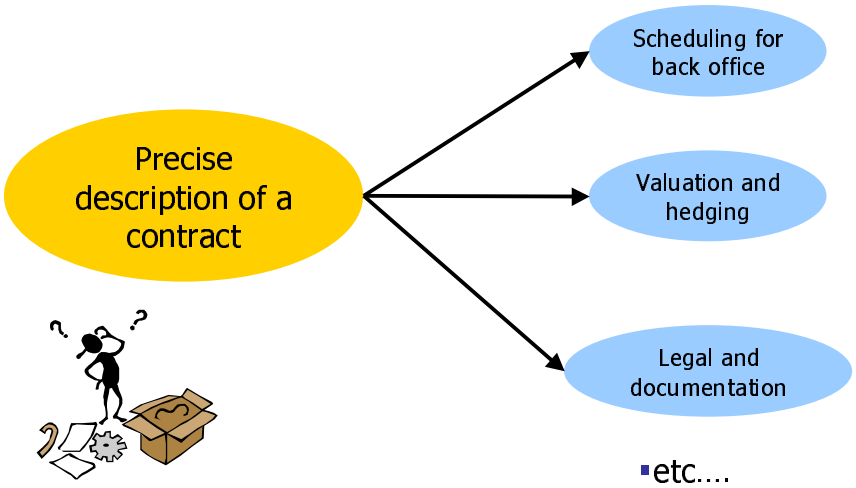


Financial contracts are complex

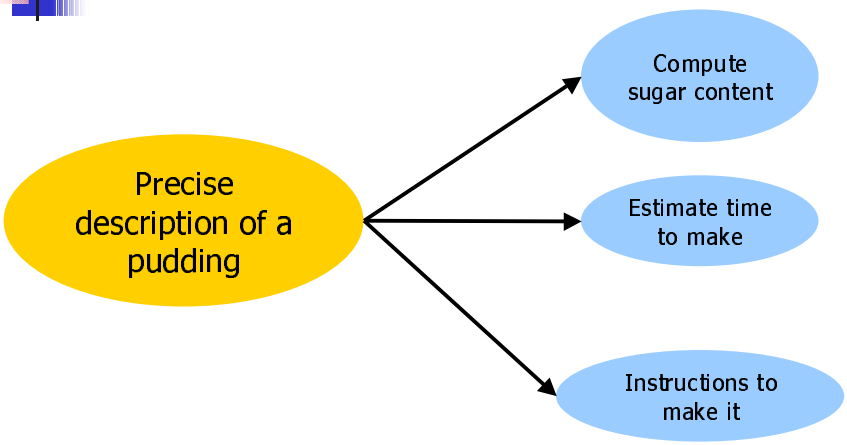
- An option, exercisable any time between t1 and t2
  - over an underlying consisting of a sequence of fixed payments
  - plus some rule about what happens if you exercise the option between payments
- plus a fixed payment at time t3

- Complex structure
- Subtle distinctions
- Need for precision

What we want to do

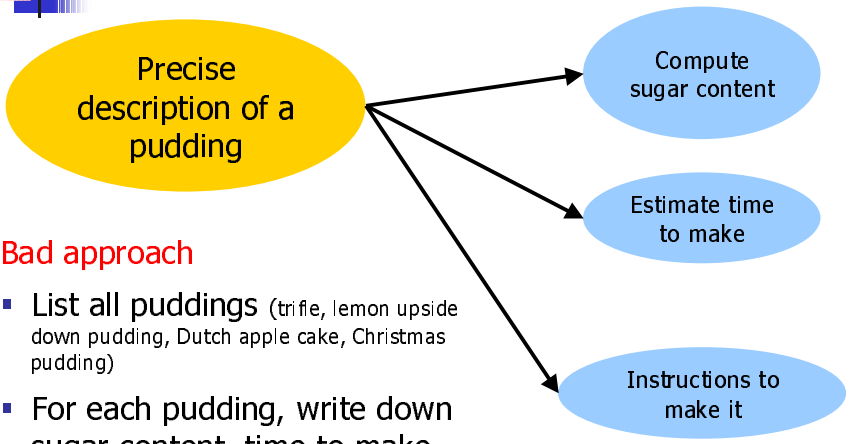


What we want to do



- etc....

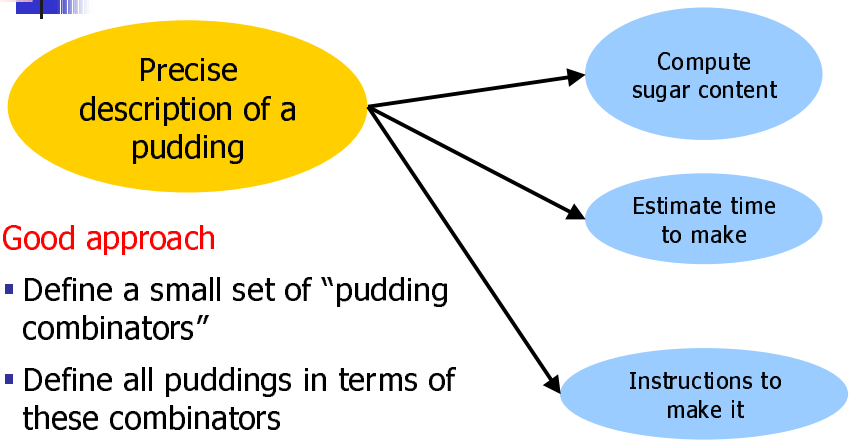
What we want to do



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

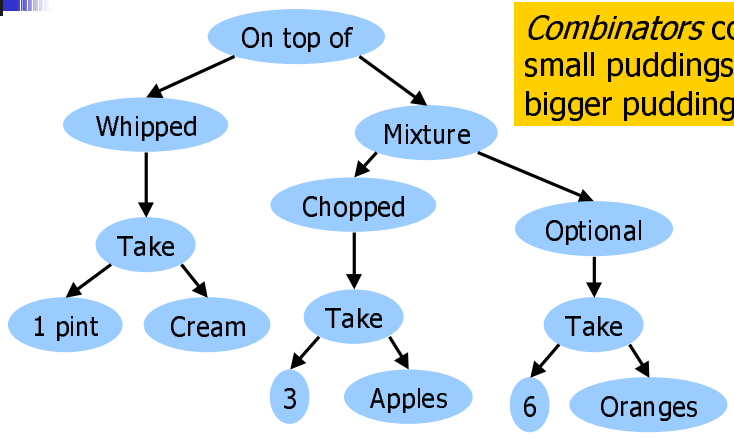
What we want to do



Good approach

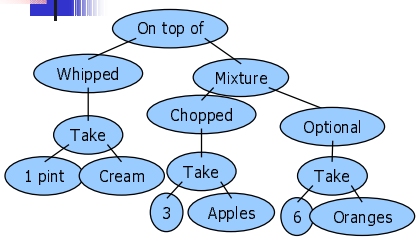
- Define a small set of "pudding combinators"
- Define all puddings in terms of these combinators
- Calculate sugar content from these combinators too

Creamy fruit salad



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

The combinators are typed

```
salad       = onTopOf topping main_part
topping     = whipped (take 1pint cream)
main_part   = mixture apple_part orange_part
apple_part  = chopped (take 3 apple)
orange_part = optional (take 6 oranges)
```

```
onTopOf :: Pudding -> Pudding -> Pudding

whipped :: Pudding -> Pudding

take :: Quantity -> Ingredient -> Pudding
```

- The types make sure that you can't describe a nonsensical (e.g. **`take topping apple`**)

Processing puddings

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

S(**onTopOf p1 p2**)	= S(**p1**) + S(**p2**)
S(**whipped p**)	= S(**p**)
S(**take q i**)	= q * 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

S(**onTopOf p1 p2**)	= S(**p1**) + S(**p2**)
S(**whipped p**)	= S(**p**)
S(take q i)	= q * S(i)
...etc...	

S is ***compositional***

To compute S for a compound pudding,

- Compute S for the sub-puddings
- Combine the results in some combinator-dependent way

Doing the same for contracts

The big question

What are the appropriate primitive combinators?

Building a simple contract

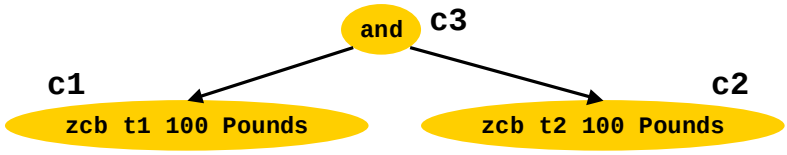
```
c1 :: Contract
c1 = zcb (date “1 Jan 2010”) 100 Pounds
```

```
zcb :: Date -> Float -> Currency -> Contract
-- Zero coupon bond

date :: String -> Date
```

Building a simple contract

```
c1,c2,c3 :: Contract
c1 = zcb (date "1 Jan 2010") 100 Pounds
c2 = zcb (date "1 Jan 2011") 100 Pounds

c3 = and c1 c2
```

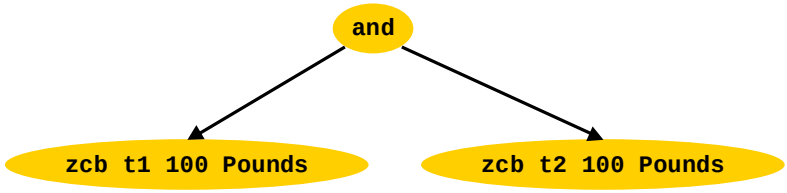
```
and :: Contract -> Contract -> Contract
-- Both c1 and c2
```



Building a simple contract

```
c3 = c1 'and' c2
```

- Notational convenience: can write combinators with two arguments in an infix position

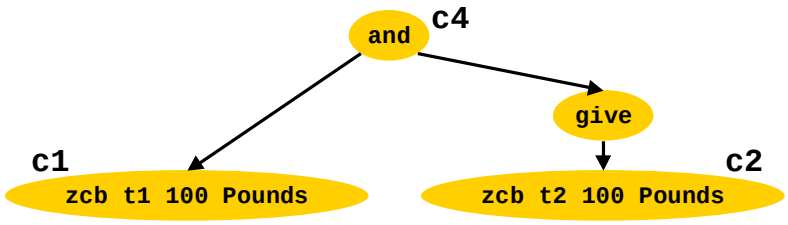


Inverting a contract

```
c4 = c1 ‘and‘ give c2
```

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

- **and** is like addition
- **give** is like negation



New combinators from old

```
andGive :: Contract -> Contract -> Contract
andGive u1 u2 = u1 ‘and‘ give u2
```

- **andGive** is a new combinator, defined in terms of simpler combinators
- To the “user” it 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

Choice

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)

Choose which

```
or :: Contract -> Contract -> Contract
-- Either c1 or c2

zero :: Contract -> Contract
-- A worthless contract that expires when
-- the underlying does
```

- First attempt at a European option

```
european :: Contract -> Contract
european u = u ‘or‘ zero u
```

- *But we need to specify when the choice may be exercised*

Acquisition dates

```
european :: Date -> Contract -> Contract
european t u = get t (u 'or' zero u)
```

- Informally, (**`get t c`**) acquires the underlying contract c at time t

```
get :: Date -> Contract -> Contract
-- Acquire the underlying at specified date
```

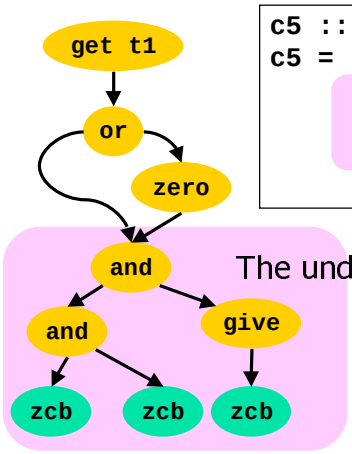
Acquisition dates

```
get :: Date -> Contract -> Contract
-- Acquire the underlying at specified date
```

- A contract confers certain **rights and obligations**
- When you acquire a contract, you take on its **future** rights and obligations; that is, only the ones that fall due on or after the acquisition date.
- If you acquire the contract `(get t u)` at time `s<t`, then you are obliged to acquire the underlying `u` at the (later) time `t`.
- If you acquire the contract `(c1 'or' c2)` you must **immediately** acquire your choice of `c1` or `c2`

Reminder…

- Remember that the underlying contract is arbitrary



The underlying

```
c5 :: Contract
c5 =  european t1 (
                zcb t2 100 Pounds
      'and'     zcb t3 100 Pounds
      'and'     give (zcb t4 200 Pounds)
      )
```

Choose when

```
anytime :: Contract -> Contract
-- Acquire the underlying at any time
-- before it expires (but you must acquire it)
```

```
c6 =  anytime (
            zcb t2 100 Pounds
      'and' zcb t3 100 Pounds
      'and' give (zcb t4 200 Pounds)
      )
```

- Every contract has a **horizon**, at which point it **expires**, and cannot be acquired

Optional acquisition

- In an American option, you can usually choose not to exercise the option at all

That's easy!

```
anytime (u 'or' zero u)
```

Choose when

Choose whether

Setting the window

An American option usually comes with a pair of times:

- you cannot acquire the underlying before t1

not quite right

```
get t1 (anytime (u 'or' zero u))
```

- you cannot acquire the underlying after t2

```
get t1 (anytime (truncate t2 (u 'or' zero u)))
```

```
truncate :: Date -> Contract -> Contract
-- You cannot acquire the underlying after
-- the specified date
```

American options

```
american :: Date -> Date -> Contract -> Contract
american t1 t2 u
  = get t1 (anytime (truncate t2 (u ‘or‘ zero u)))
```

Extensible library

Combinators

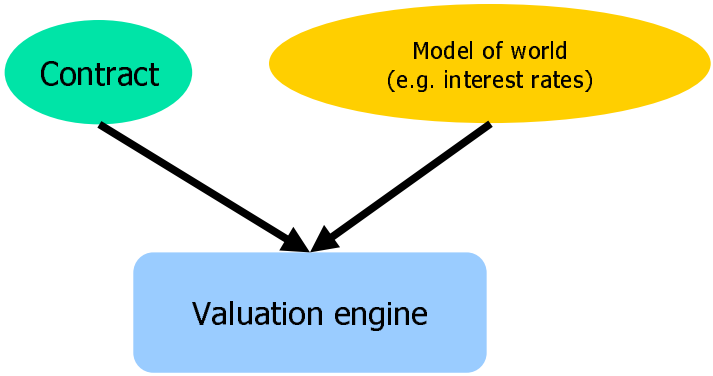
Summary so far

```
give       :: Contract -> Contract
or         :: Contract -> Contract -> Contract
and        :: Contract -> Contract -> Contract
zero       :: Contract -> Contract
get        :: Date -> Contract -> Contract
anytime    :: Contract -> Contract
truncate   :: Date -> Contract -> Contract
…and some more besides…
```

- Choice of combinators driven by
  - Economy (as few as possible)
  - Expressiveness (can describe many contracts)
  - Efficiency (maps cleanly onto e.g. valuation engine)
- We need an absolutely precise specification of what they mean

Valuation

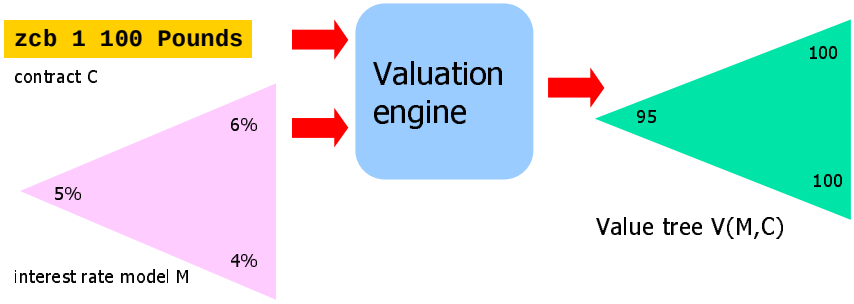
- Once we have a precise contract specification, we may want to value it



One possible evaluation model: BDT

Given a contract C, define

V(M,C) to be the BDT tree for C under interest rate model M



Compositional valuation

Now define V(M,C) compositionally

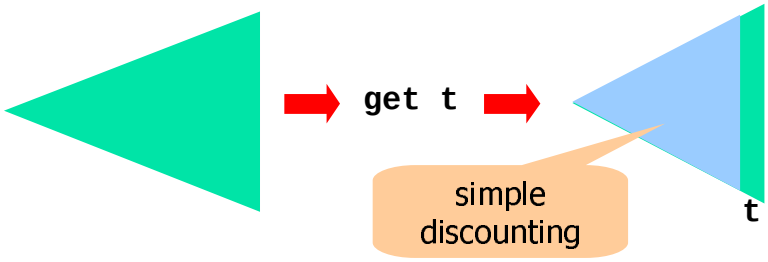
Add value trees point-wise

V(M, **c1 'and' c2**) = V(M,**c1**) + V(M,**c2**)
V(M, **c1 'or' c2**) = max(V(M,**c1**), V(M,**c2**))
V(M, **give c**) = - V(M,**c**)
V(M, **anytime c**) = snell(V(M,**c**))
V(M, **get t c**) = discount(V(M,**c**)[t])
…etc…

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

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



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"

Reasoning about contracts

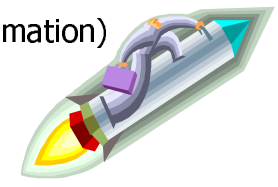
- Two contracts may **look** different, but **be** the same

```
c1 'and' c2  =  c2 'and' c1
```

- We add a set of rules about equality to our language
- Using these rules we can transform a contract into an equivalent one that takes less work to evaluate

```
anytime (anytime c) = anytime c
```

(cf: query optimisation, program transformation)



Summary

- A small set of built-in combinators
- A user-extensible library defines the zoo of contracts
- So you can define an infinite family of contracts

- Compositional (modular) algorithms for valuation, and other purposes
- Not covered: observables. See the paper.

- Prototype implementation in Haskell. (100's not 1000's of lines).