Based on slides by Simon Peyton-Jones, Julian Seward, Jean-Marc Seber

DOMAIN-SPECIFIC LANGUAGES

58

58

Domain-Specific Languages

- Encoding domain expertise
  - Domain abstractions
  - Operations for manipulating abstractions
- Application generators
  - Can customization of generated app be related to original DSL program?
- Prototyping languages
  - E.g. evaluate financial instruments before production implementation

59

59

- RISLA:
  - Standalone domain-specific language
  - Rapid prototyping of financial instruments
  - Object-oriented
  - Compiled to COBOL

```
product LOAN
declaration
  contract data
    PAMOUNT : amount                %% Principal Amount
    STARTDATE : date                %% Starting date
    MATURDATE : date                %% Maturity data
    INTRATE : int-rate              %% Interest rate
    RDMLIST := [] :  cashflow-list  %% List of redemptions.
  information
    PAF : cashflow-list             %% Principal Amount Flow
    IAF : cashflow-list             %% Interest Amount Flow
  registration
    %% Register one redemption.
    RDM(AMOUNT : amount, DATE : date)
  local
    %% Final Principal Amount
    FPA(CHFLLIST : cashflow-list) :  amount
    %% Final redemption
    FRDM : cashflow
  error checks
    "Wrong term dates" in case of STARTDATE >= MATURDATE
    "Negative amount" in case of PAMOUNT < 0.0
implementation
  local
    define FPA as IBD(CHFLLIST, -/-PAMOUNT, MATURDATE)
    define FRDM as <-/-FPA(RDMLIST), MATURDATE>
  information
    define PAF as [<-/-PAMOUNT, STARTDATE>] >> RDMLIST >> [FRDM]
    define IAF as [< -/-CIA( BL(RDMLIST, <-/-PAMOUNT,
                               <STARTDATE, MATURDATE>>),
                             INTRATE ), MATURDATE >]
  registration
    define RDM as
      error checks
        "Date not in interval" in case of (DATUM < STARTDATE)
                               or (DATUM >= MATURDATE)
        "Negative amount"      in case of AMOUNT <= 0.0
        "Amount too big"       in case of
            FPA(RDMLIST >> [<AMOUNT, DATE>]) > 0.0
      RDMLIST := RDMLIST >> [<AMOUNT, DATE>]
```


- Haskell Contracts
  - Embedded domain-specific language
  - Communication among domain experts
  - Declarative
  - Embedded in Haskell functional language

`zero :: Contract`
"`zero` is a contract that may be acquired at any time. It has no rights and no obligations, and has an infinite horizon."

`one :: Currency → Contract`
"`(one k)` is a contract that immediately pays the holder one unit of the currency `k`. The contract has an infinite horizon."

`give :: Contract → Contract`
"To acquire `(give c)` is to acquire all of `c`'s rights as obligations, and vice versa. For a bilateral contract `q` between parties *A* and *B*, *A* acquiring `q` implies that *B* acquires `(give q)`."

`and :: Contract → Contract → Contract`
"If you acquire `c1 'and' c2)` then you immediately acquire both `c1` (unless it has expired) and `c2` (unless it has expired). The composite contract expires when both `c1` and `c2` expire."

`or :: Contract → Contract → Contract`
"If you acquire `c1 'and' c2)` then you immediately acquire either `c1` or `c2` (but not both). If either has expired, that one cannot be chosen. When both have expired, the compound contract expires."

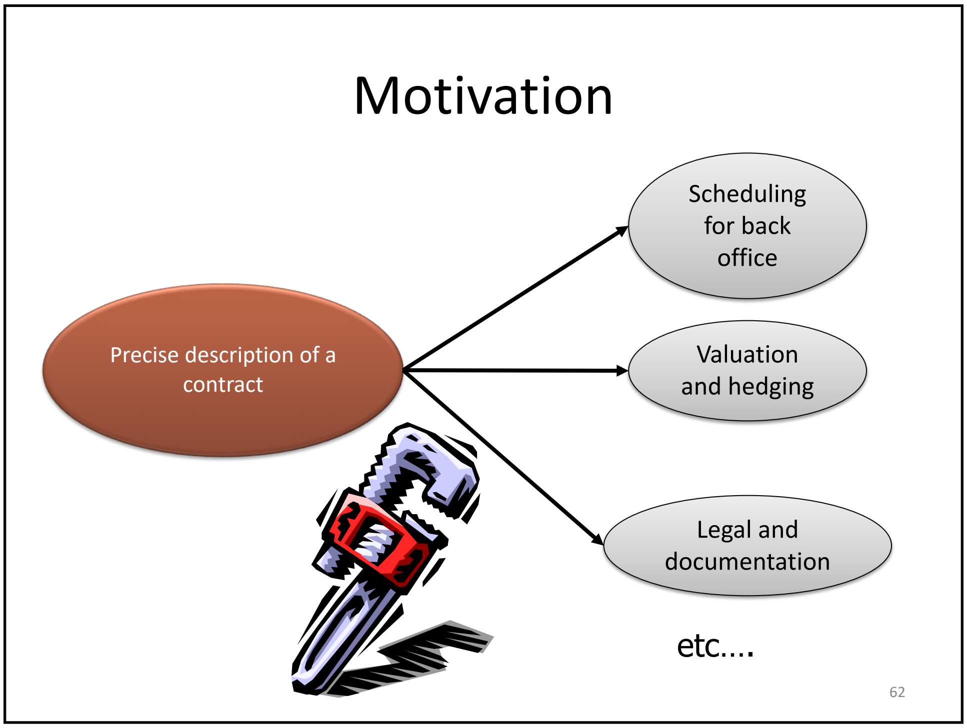
`truncate :: Date → Contract → Contract`
"`(truncate t c)` is exactly like `c` except that it expires at the earlier of `t` and the horizon of `c`. Notice that `truncate` limits only the possible *acquisition date* of `c`; it does not truncate `c`'s rights and obligations, whhich may extend well beyond `t`."

`then :: Contract → Contract → Contract`
"If you acquire `(c1 'then' c2)` and `c1` has not expired, then you acquire `c1`. If `c1` has expired, but `c2` has not, then you acquire `c2`. The compound contract expires when both `c1` and `c2` expire."

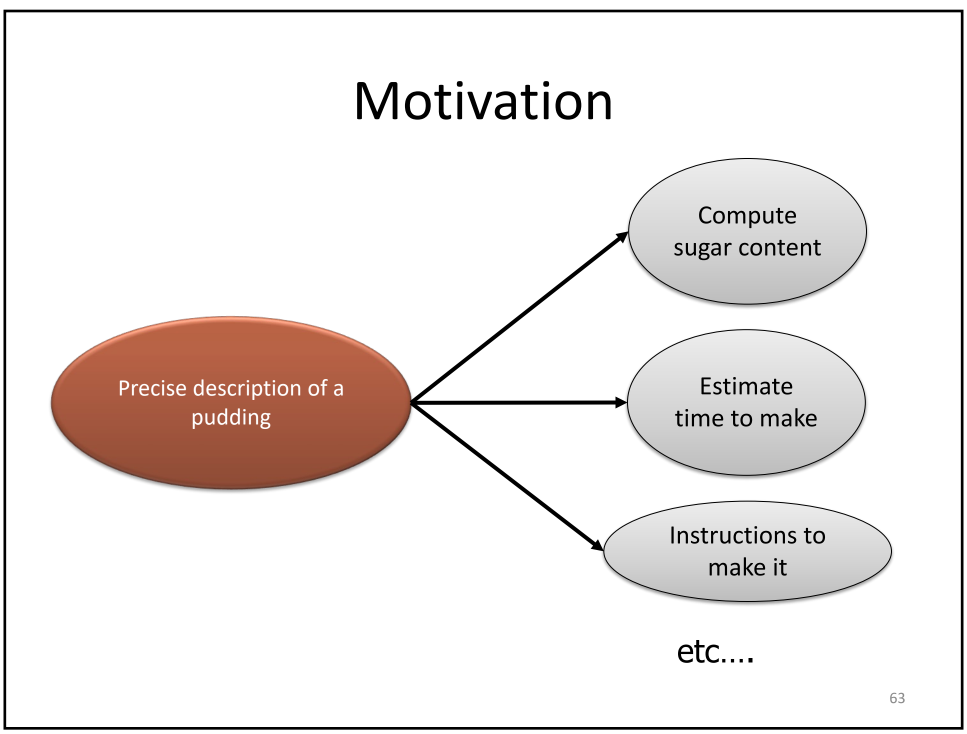
`scale :: Obs Double → Contract → Contract`
"If you acquire `(scale o c)`, then you acquire `c` at the same moment, except that all the rights and obligations of `c` are multiplied by the value of the observables `o` at the moment of acquisition."

`get :: Contract → Contract`
"If you acquire `(get c)` then you must acquire `c` at `c`'s expiry date. The compound contract expires at the same moment that `c` expires."

`anytime :: Contract → Contract`
"If you acquire `(anytime c)` you must acquire `c`, but you can do so at any time between the acquisition of `(anytime c)` and the expiry of `c`. The compound contract expires when `c` does."

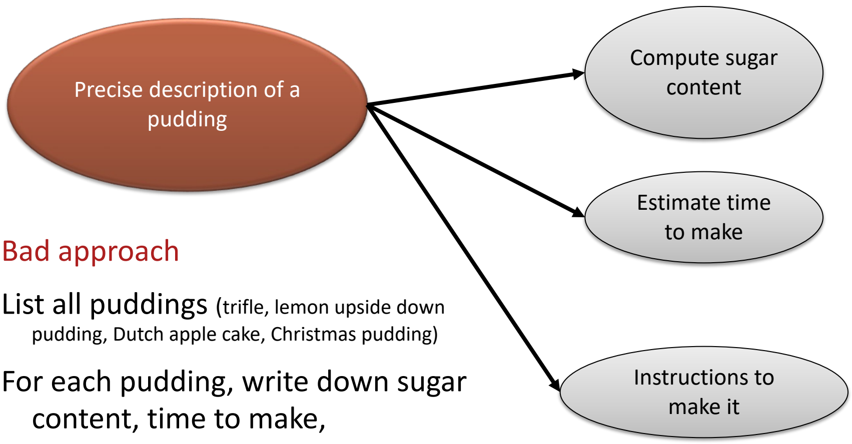

Motivation
Precise description of a contract
Scheduling for back office
Valuation and hedging
Legal and documentation
etc….
62

62


Motivation
Precise description of a pudding
Compute sugar content
Estimate time to make
Instructions to make it
etc….
63

63

Motivation



Bad approach

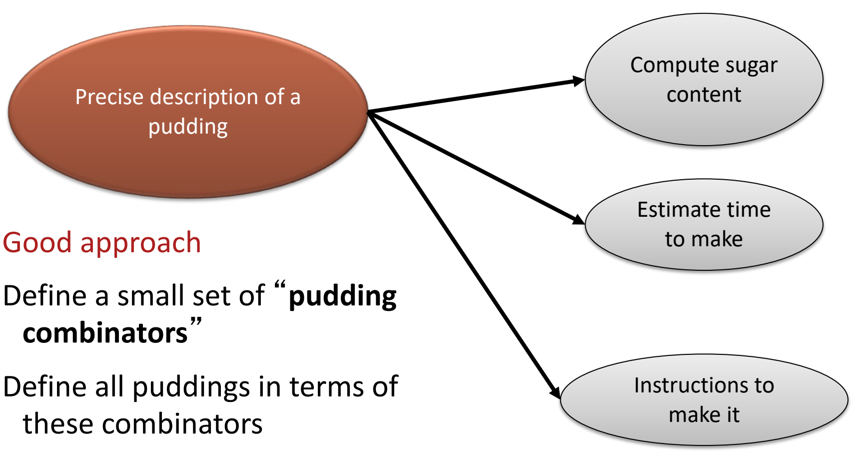
List all puddings (trifle, lemon upside down pudding, Dutch apple cake, Christmas pudding)

For each pudding, write down sugar content, time to make, instructions etc

64

64

Motivation



Good approach

Define a small set of "**pudding combinators**"

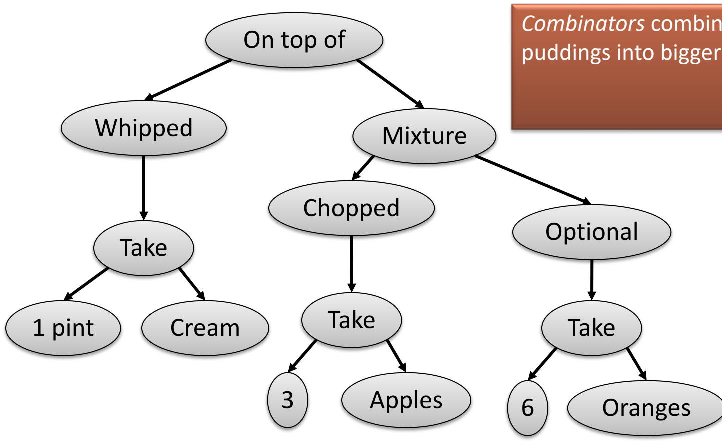
Define all puddings in terms of these combinators

Calculate sugar content from these combinators too

65

65

Creamy fruit salad

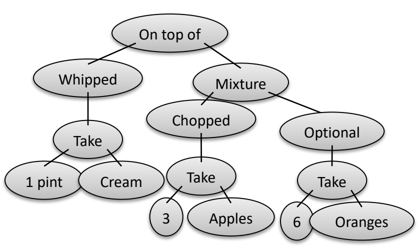


Combinators combine small puddings into bigger puddings

66

66

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

67

67

Building a simple contract

"Receive $100 on 1 Jan 2020"

```
c1 :: Contract
c1 = zcb (date "1 Jan 2020") 100 Dollars
```

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

Combinators will appear in gold boxes

68

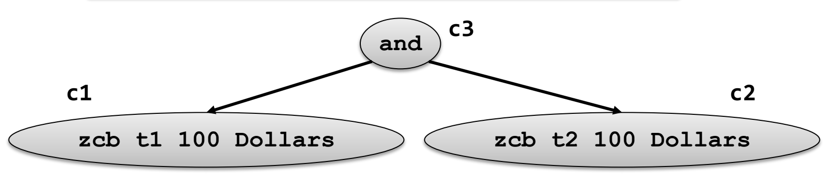
68

Combining contracts

```
c1,c2,c3 :: Contract
c1 = zcb (date "1 Jan 2020") 100 Dollars
c2 = zcb (date "1 Jan 2021") 100 Dollars

c3 = and c1 c2
```

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



69

69

Inverting a contract

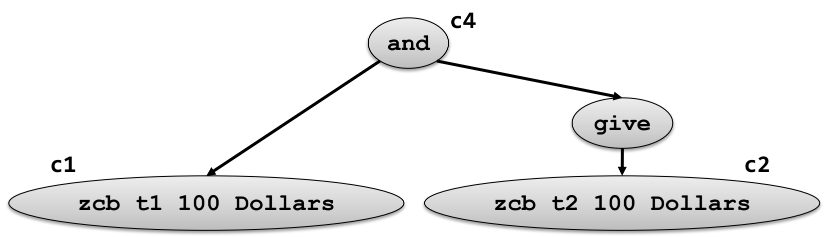
Backquotes for infix notation

```
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”, **andGive** is no different from 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**

Defining zcb

Indeed, **zcb** is not primitive:

```
zcb :: Date → Float → Currency → Contract
                                 (one Dollar)
```

```
one :: Currency → Contract
-- Receive one unit of currency immediately
```

72

72

Defining zcb

Indeed, **zcb** is not primitive:

```
zcb :: Date → Float → Currency → Contract
                         (scaleK 100 (one Dollar))
```

```
one :: Currency → Contract
-- Receive one unit of currency immediately

scaleK :: Float → Contract → Contract
-- Acquire specified number of contracts
```

73

73

Defining zcb

Indeed, **zcb** is not primitive:

```
zcb :: Date → Float → Currency → Contract
at (Date “1 Jan 2020”) (scaleK 100 (one Dollar))
```

```
one :: Currency → Contract
-- Receive one unit of currency immediately

scaleK :: Float → Contract → Contract
-- Acquire specified number of contracts

at :: Date → Contract → Contract
-- Acquire the contract at specified date
```

74

74

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 **(one k)**, you receive one unit of currency **k immediately**

If you acquire the contract **(at t u)** at time **s<t**, then you acquire the contract **u** at the (later) time **t**.

You cannot acquire **(at t u)** after **t**. The latest acquisition date of a contract is its **horizon**.

75

75

Choice

- An *option* gives the flexibility to:

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

- **Choose when** to acquire a contract
  - exercising the option = acquiring the underlying contract

76

76

Choose **which**

European option: at a particular date you may choose to acquire an "underlying" contract, or to decline

```
european :: Date → Contract → Contract
european t u = at t (u `or` zero)
```

```
or :: Contract → Contract → Contract
-- Acquire either c1 or c2 immediately

zero :: Contract
-- A worthless contract
```

77

77

Choose **when**: American options

The option to acquire 10 Microsoft shares, for $100, anytime between t1 and t2 years from now

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

anytime: Choose *when*

```
golden_handcuff = anytime shares

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

or: Choose *whether*

MS shares are a "currency"

78

Setting the window

```
golden_handcuff = (anytime shares)
```

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

79

79

Setting the window

```
golden_handcuff = (anytime (truncate t2 shares))
```

Can't acquire shares after t2

```
truncate :: Date → Contract → Contract
-- Truncate the horizon of a contract
```

80

80

Setting the window

```
golden_handcuff = at t1
                    (anytime (truncate t2 shares))
```

Acquire the anytime rights at t1

Can't acquire shares after t2

```
truncate :: Date → Contract → Contract
-- Truncate the horizon of a contract

at :: Date → Contract → Contract
-- Acquire the underlying contract at specified date
```

81

81

Observables

Pay me $1000 * (the number of inches of snow - 10) on 1 Jan 2022

```
c :: Contract
c = at "1 Jan 2022" (scale scale_factor (one Dollar))

scale_factor :: Observable
scale_factor = 1000 * (snow - 10)
```

```
scale :: Observable → Contract → Contract
-- Scale the contract by the value of the observable
-- at the moment of acquisition

snow :: Observable
(*), (-) :: Observable → Observable → Observable
```

82

82

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

83

83

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

84

84

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 results in some combinator-dependent way

85

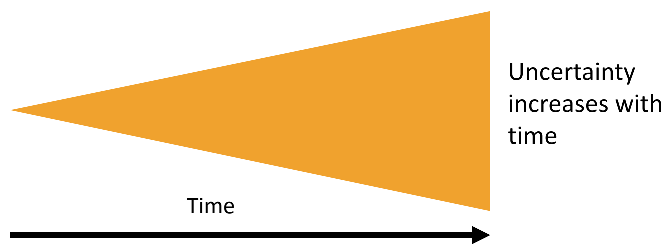
85

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.

$\mathcal{E}$: Contract → RandomProcess

RandomProcess = Time → RandomVariable



Compositional valuation

Add random processes point-wise

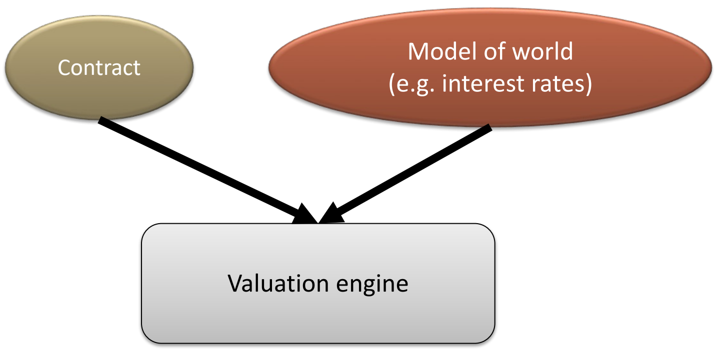
```
Ɛ(c1 `and` c2)      = Ɛ (c1) + Ɛ (c2)
Ɛ(c1 `or` c2)       = max(Ɛ(c1), Ɛ(c2) )
Ɛ(give c)           = - Ɛ (c)
Ɛ(anytime c)        = snell( Ɛ(c) )
Ɛ(at t c)           = discount( Ɛ(c)[t] )
```

…etc…

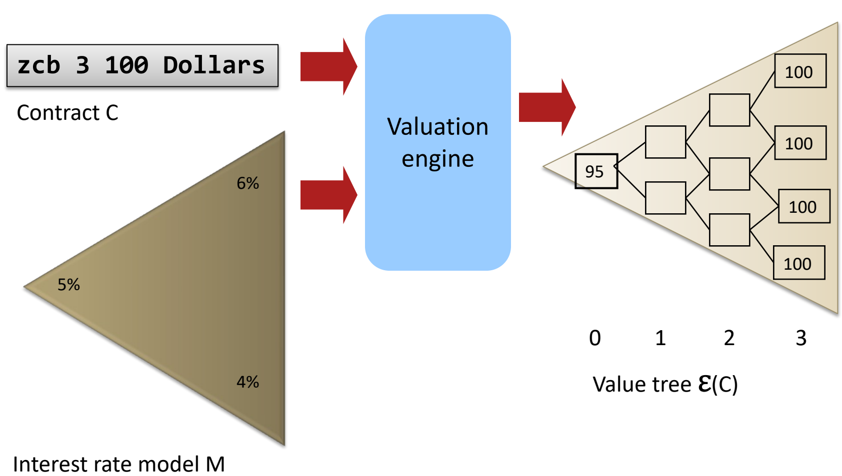
This is a **major payoff**! Deal with the 10-ish combinators, and we are done with valuation!

Valuation

- There are many numerical methods to compute discrete approximations to random processes



Example Evaluation Model: BDT



Reasoning about equivalence

- Using this semantics we can prove (for example) that

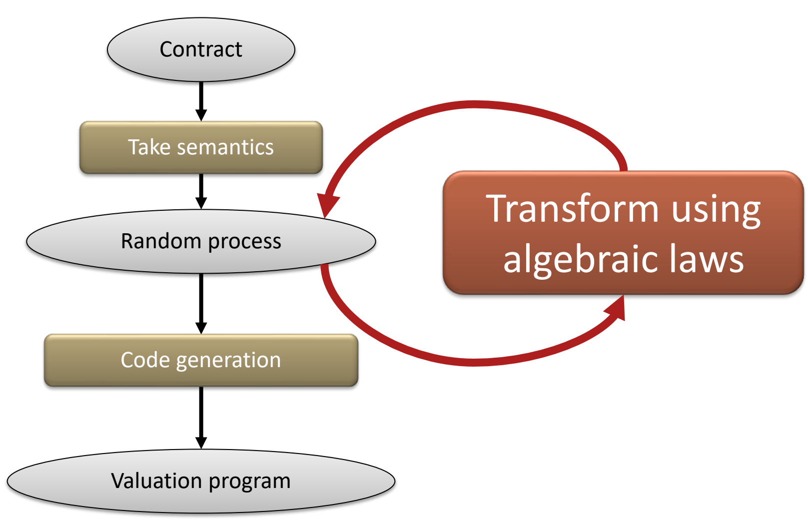
```
anytime (anytime c) = anytime c
```

- Depends on algebra of random processes (snell, discount, etc).

90

90

A compiler for contracts



91

91