

# Compositional Description, Valuation, and Management of Financial Contracts: The MLFi Language

IIª Jornada de Riesgos Financieros

RiskLab-Madrid

November 14, 2002

Jean-Marc Eber, LexiFi

jeanmarc.eber@lexifi.com

*Based on joint theoretical work by*

*Simon Peyton Jones, Microsoft Research, Cambridge*

*Pierre Weis, INRIA and LexiFi, Paris and*

*Jean-Marc Eber, LexiFi, Paris*

# Contents

- ◆ The Challenge
- ◆ Compositional Description of Financial Contracts
- ◆ Valuation
- ◆ Management
- ◆ Other Applications
- ◆ Summary

# The Challenge

# What We Hear and See

“Self-explaining data”

“B2B automation”

“Multiple, new products”

“Link 3rd party and proprietary systems”

“Disparate users”

“Operational risk”

“Flexibility... but control”

“Standards”

“Reusability”

“Aggregation”

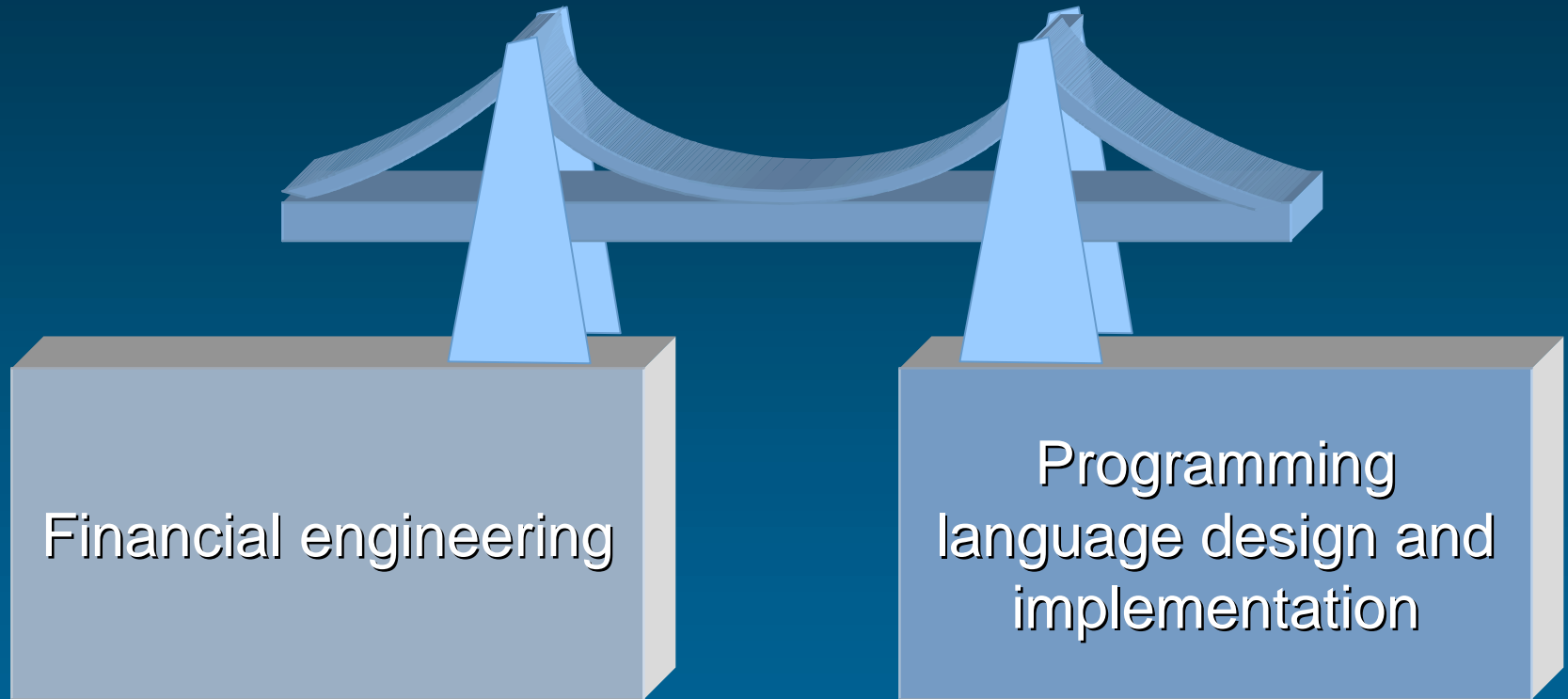
“Extensible”

“Formal OTC exchanges”

# Business Drivers

- ◆ Growing number and complexity of products
  - ❖ that require increasingly precise descriptions
- ◆ New and evolving processes
- ◆ Mastering operational risk as a source of competitive advantage for any financial institution
- ◆ Unified approach of insurance, finance and "retail" contracts
- ◆ Regulation
- ◆ Importance of cross-functional processes (VaR, regulatory reporting, Raroc, credit risk)
- ◆ B2B "automation"

# The Big Picture



# Compositional Description of Financial Contracts

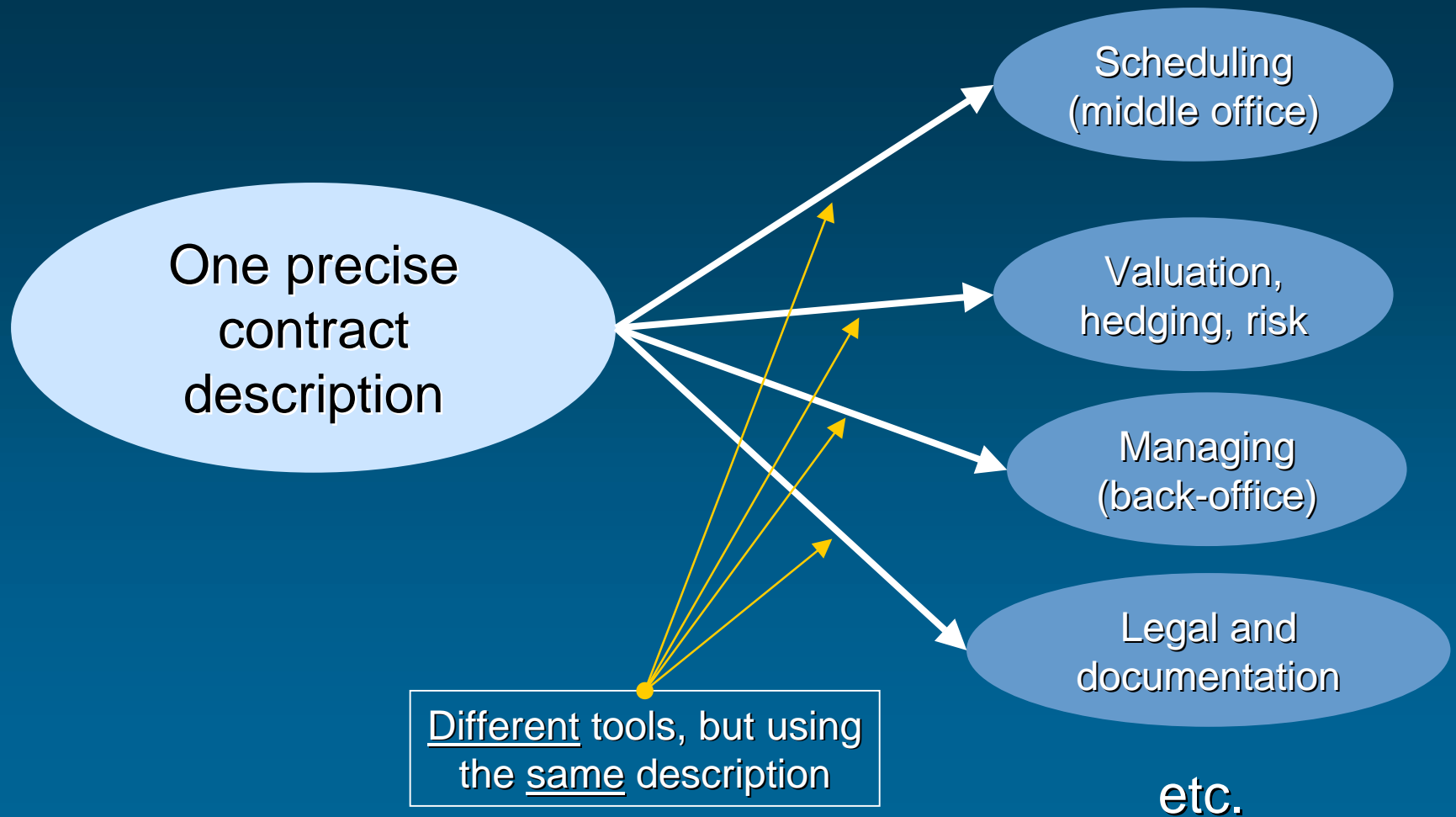
# Financial Contracts Are Complex

Example:

- ◆ An option, exercisable any time between  $t_1$  and  $t_2$ 
  - ❖ on 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  $t_3$

- ◆ Complex structure
- ◆ Subtle distinctions
- ◆ Need for precision  
...for many uses!

# What We Want to Do

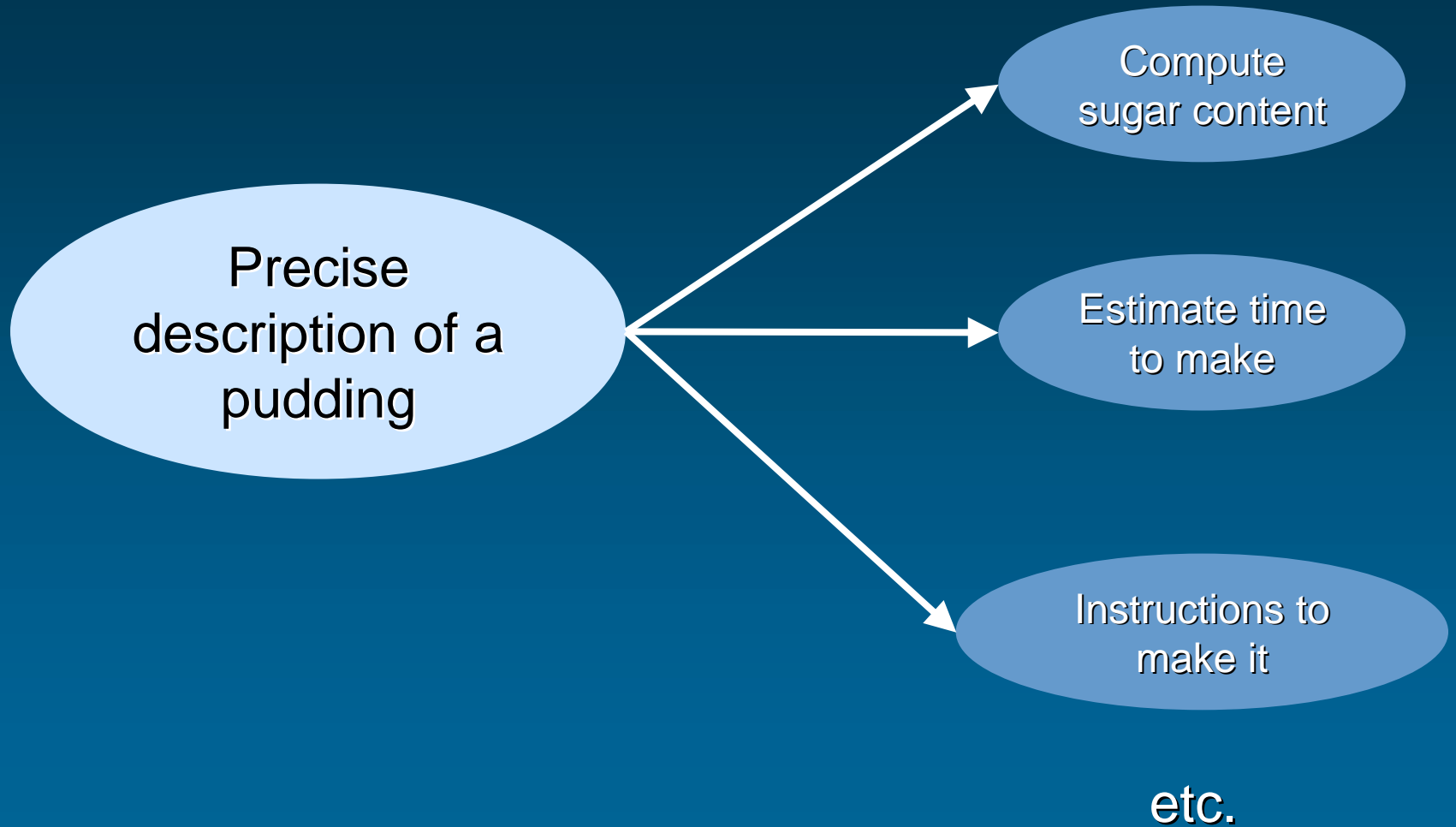


# What We Want to Do (2)

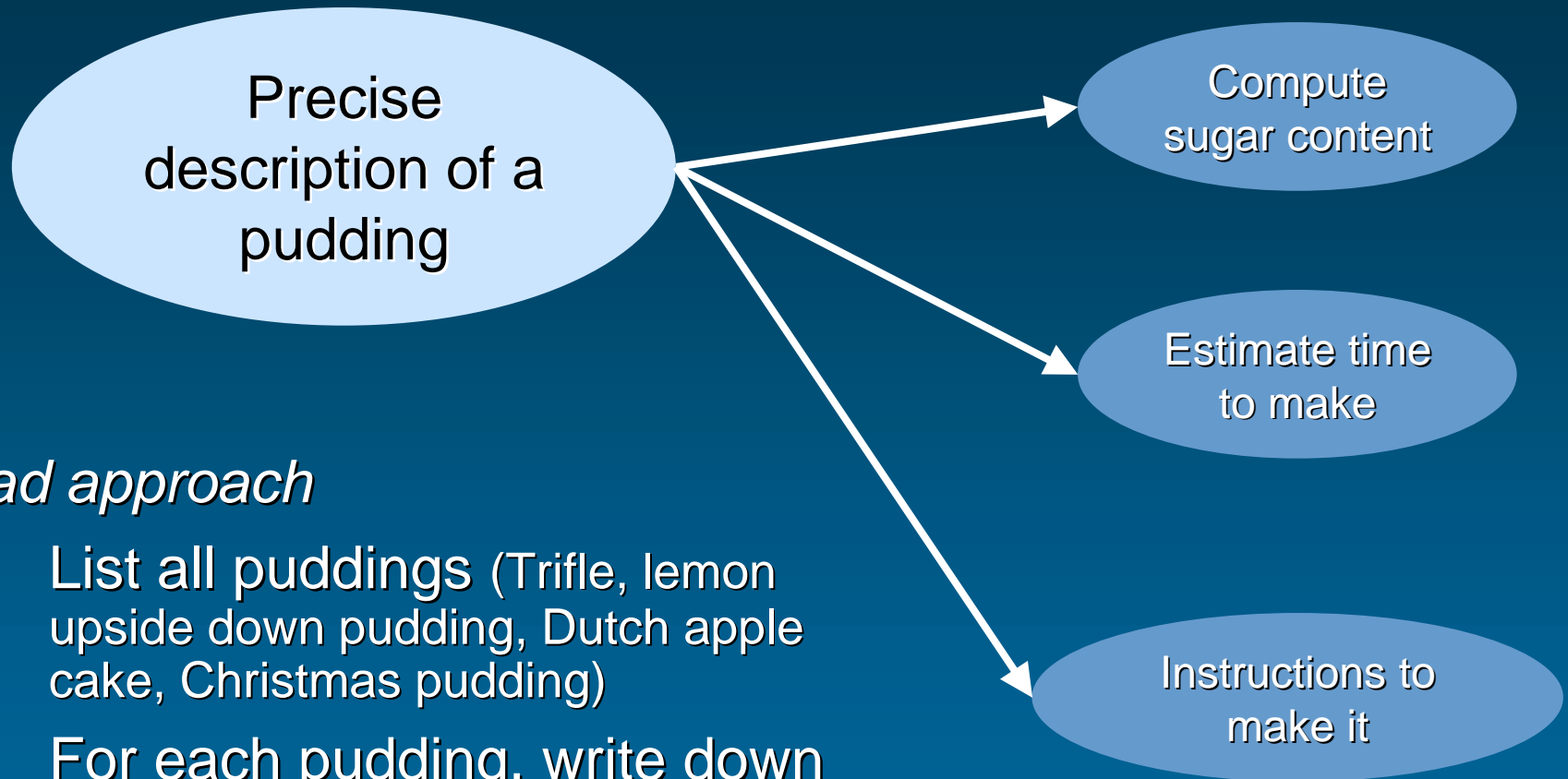
Precise description means:

- ◆ A syntax (easy), may be a “language”, a graphical notation, etc. that combines some elementary "atoms" with "operators"
- ◆ A clear semantics (difficult): what do the operators *mean*?
- ◆ But one unambiguous description shared between different people and different applications

# An Analogy



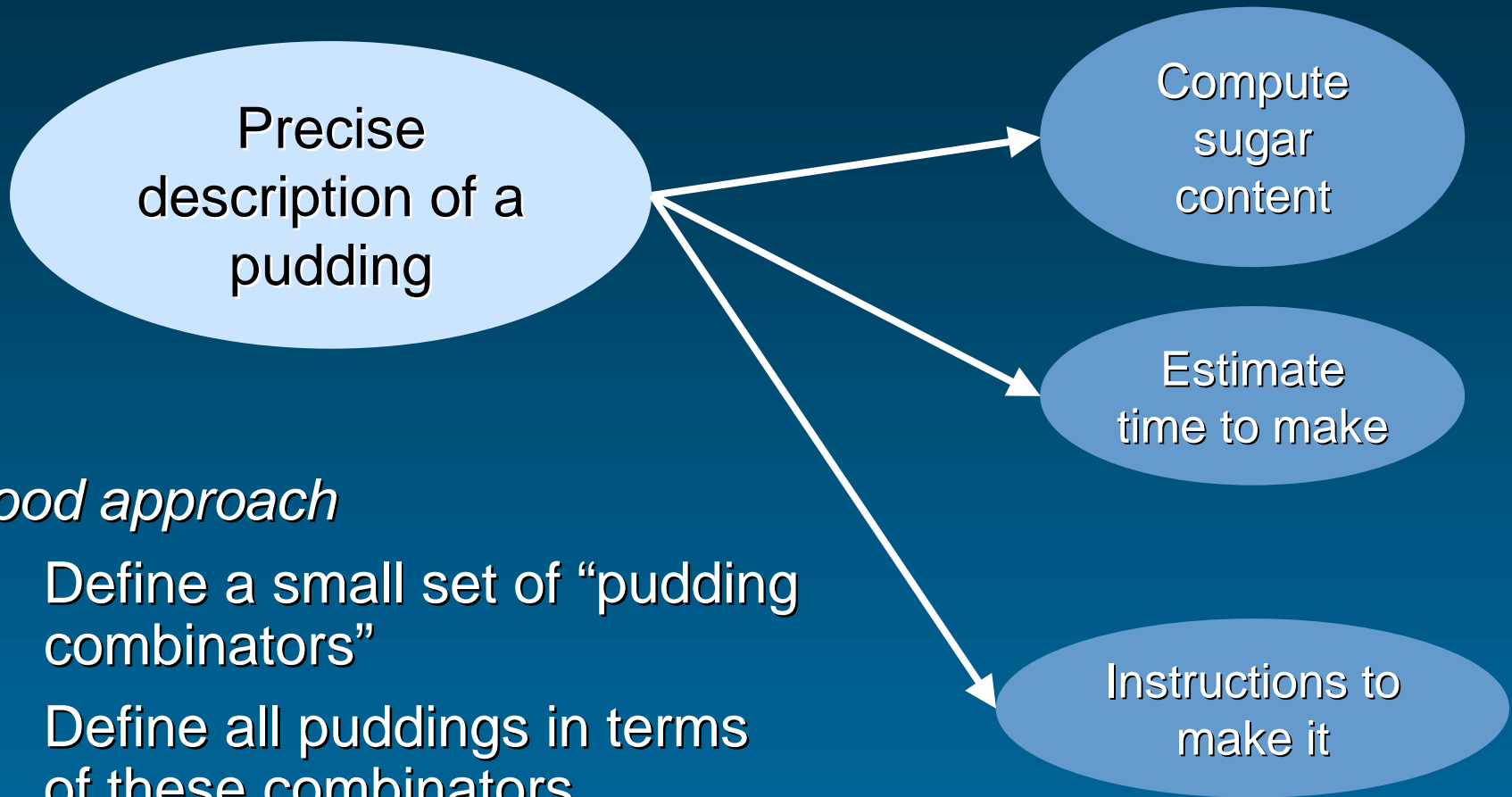
# An Analogy (2)



## *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

# An Analogy (3)

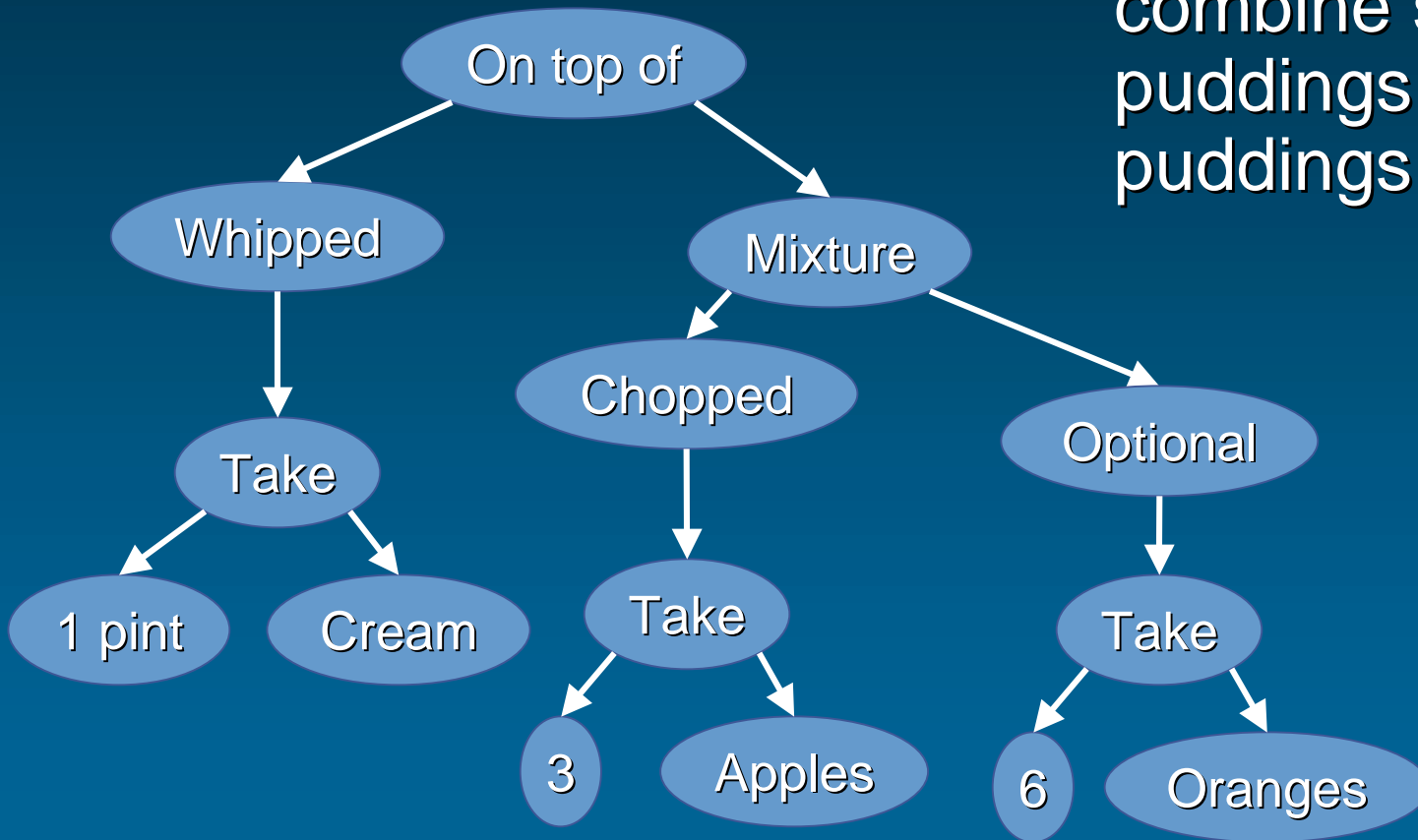


## *Good approach*

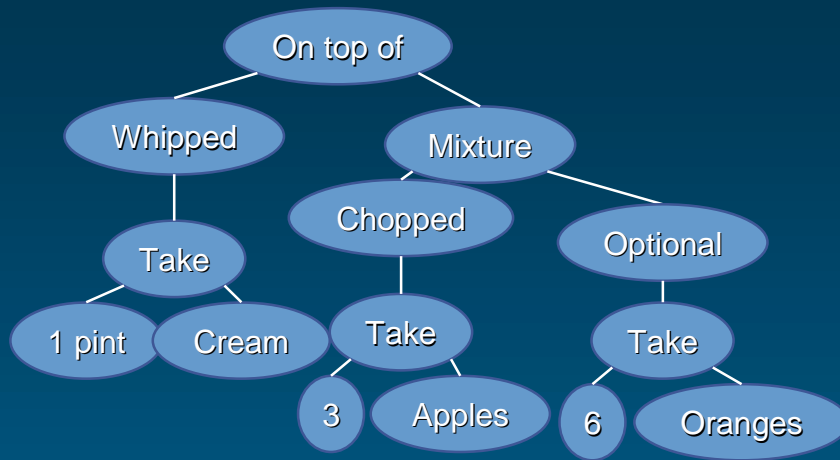
- ◆ Define a small set of “pudding combinators”
- ◆ Define all puddings in terms of these combinators
- ◆ Calculate sugar content from these combinators too

# 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

# Processing Puddings

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

$$S(\text{onTopOf } p1 \ p2) = S(p1) + S(p2)$$

$$S(\text{whipped } p) = S(p)$$

$$S(\text{take } q \ i) = q * S(i)$$

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 would we need to enhance  $S$

# Processing Puddings (2)

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

$$S(\text{onTopOf } p1 \ p2) = S(p1) + S(p2)$$

$$S(\text{whipped } p) = S(p)$$

$$S(\text{take } q \ i) = q * S(i)$$

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 for financial contracts?

- ◆ How to achieve a compositional approach?
- ◆ How to master the different uses in one description?
- ◆ Build a **theory of financial contract description**

# Building a Simple Contract

```
c1 : contract
c1 = zcb(2010-01-25, 100, GBP)
```

```
zcb : (date * float * currency) -> contract
(* Zero coupon bond *)
```

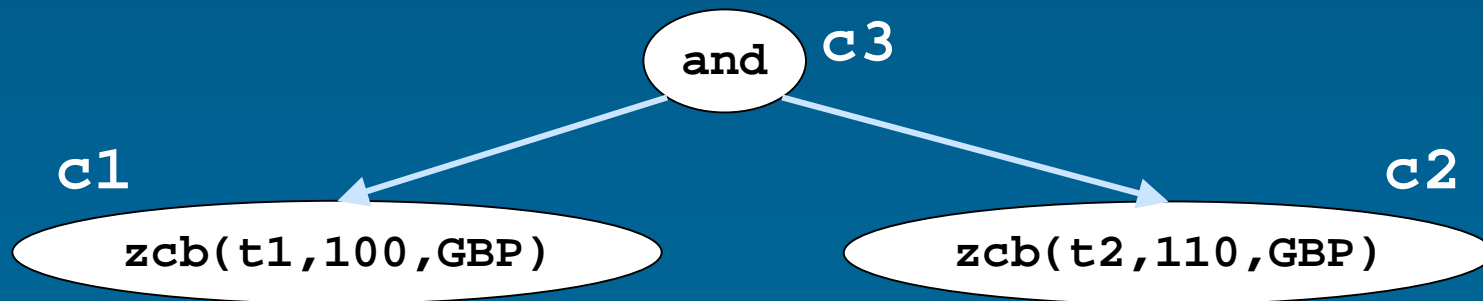
- ◆ `contract` is a build-in type of our language (like `integer` in a classical programming language) !
- ◆ `date` is also a build-in type (and `2010-01-25` is recognized as a constant of this type, like `78.5` is recognized as a `float`).

# Building a Simple Contract (2)

```
c1,c2,c3 : contract
c1 = zcb(2010-01-25, 100, GBP)
c2 = zcb(2011-01-24, 110, GBP)

c3 = and(c1, c2)
```

```
and : (contract * contract) -> contract
(* Both c1 and c2 *)
```



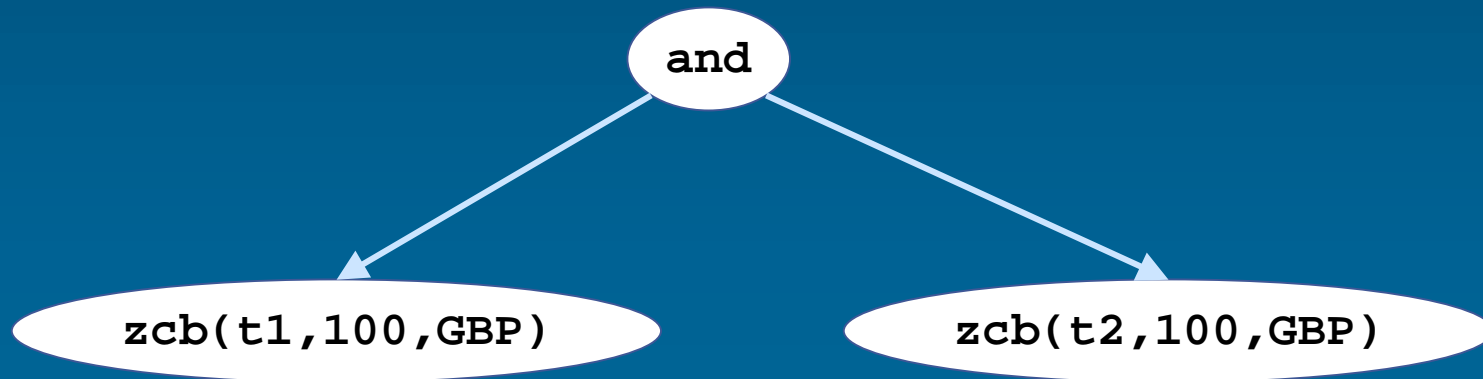
# Building a Simple Contract (3)

```
c3 = and(c1, c2)
```

*can be written*

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

Notational convenience: write combinators with two arguments in an infix position (like 'and')

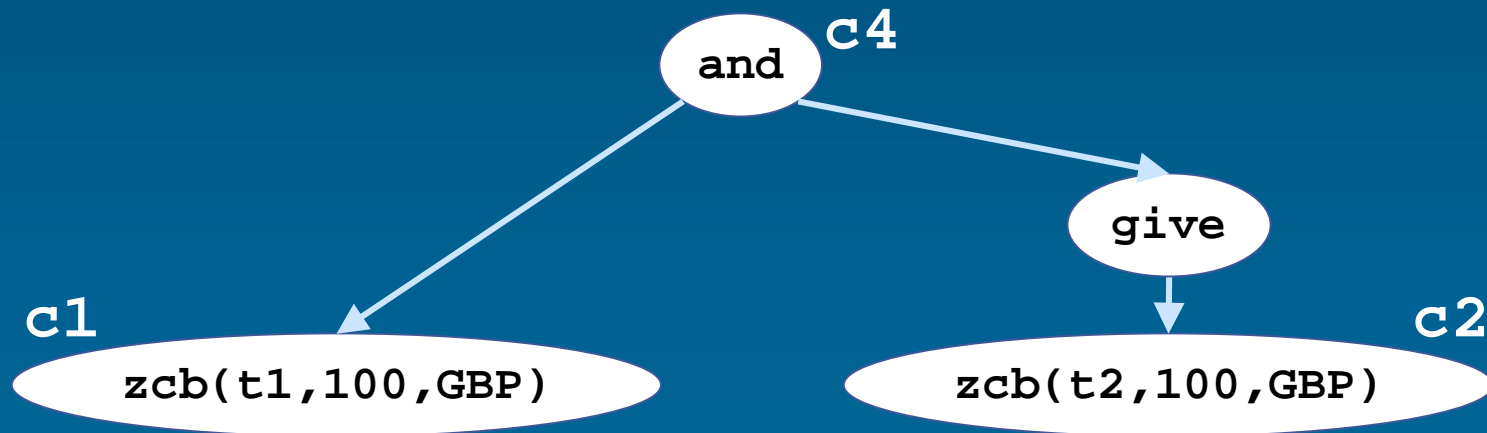


# Inverting a Contract

$c4 = c1 \text{ 'and' } (\text{give } c2)$

`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 not 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

# Choice

An option gives the holder 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
(* A contract without any right or
obligation *)
```

- ◆ First attempt at a European option

```
european : contract -> contract
european u = u 'or' zero
```

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

# Temporal Acquisition of Contracts

```
acquire : (region * contract) -> contract
(* Obligation, the first time the region is
hit, to acquire the argument contract. *)
```

A region can be complex. But here we consider a very simple one:  $\{ [t] \}$  is the region "date t"

```
european : (date * contract) -> contract
european(t, u) = acquire({[t]}, u 'or' zero)
```

- ◆ A contract is *acquired* on its *acquisition* date
- ◆ If you acquire the contract (c1 'or' c2) you must *immediately* choose which of c1 or c2 to acquire

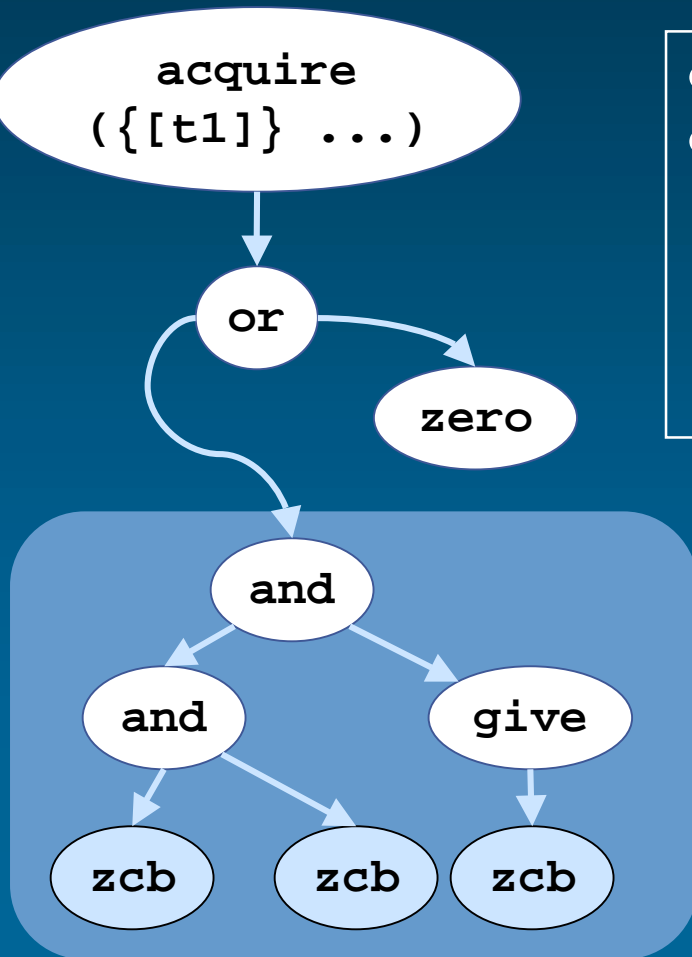
# Pragmatics: The Horizon

- ◆ Each contract  $c$  has a *horizon*  $H(c)$ , the latest possible date on which it may be *acquired*.
- ◆ A violation of this rule (that is, a potential acquisition after the horizon) is flagged as an *error* by the MLFi compiler
- ◆ Traps most of the typical contract design errors
- ◆ Each region  $r$  has a horizon  $h(r)$ , the latest possible date on which it may be true
- ◆ The horizon of a contract is defined compositionally:

```
H(c1 +and' c2) = min(H(c1), H(c2))
H(give c)      = H(c)
H(acquire(r, c)) = h(r)
                ...
```

# Reminder

Remember that the underlying contract is *arbitrary*



```
c5 : contract
c5 = european(t1, (
                zcb(t2,100,GBP)
                'and' zcb(t3,100,GBP)
                'and' give (zcb(t4,200,GBP))
                ))
```

The underlying

# Observables

- ◆ Any (often uncertain) future value on which both contract parties will agree at realization date
- ◆ Observables are observed at a date
- ◆ Contracts are written “on” observables
  - ❖ e.g., deliver a cash amount at date  $t$  depending on an observed interest rate at this date

# Observables (2)

- ◆ May be a number (e.g., a quoted stock price), a boolean (e.g., default, non default), or something else (e.g., a credit class)
- ◆ Observables may be manipulated in many ways to obtain new observables
  - ❖  *$f(o_1, o_2, \dots)$  is an observable if the  $o_i$ 's are*
  - ❖ *an observable may be equal to an observable observed 10 days earlier*
  - ❖ *etc.*

# Acquisition Date and Horizon

- ◆ The MLFi language describes what a contract *is*
  - ◆ However, the *consequences for the holder* of the contract depend on the contract's *acquisition date*
- 
- ◆ The horizon of a contract is a (compositional) property of the contract
  - ◆ The acquisition date is not!

# Choose When...

```
anytime : (region * contract * contract) -> contract
(* Acquire immediately the first contract and
   have the right, at any time in the argument region,
   to "give it back" against acquiring
   the second contract *)
```

Any payment (in or out) due *before* acquisition date  
is simply *discarded*

# ... and Setting the Window

Let's use here a slightly more complex region than in our previous example: the time interval, noted  $\{ [ t_1, t_2 ] \}$ , "from  $t_1$  to  $t_2$ "

Note that former  $\{[t]\}$  is a shorthand for  $\{[t, t]\}$

An American option usually comes with a pair of times:

- ◆ You cannot acquire the underlying  $u$  *before*  $t_1$ ;
- ◆ You cannot acquire the underlying  $u$  *after*  $t_2$ ;

```
anytime( { [t1, t2] }, zero, u )  
(* right to acquire u between t1 and t2 *)
```

# American Options

```
american : (date * date * contract) -> contract  
american(t1, t2, u) =  
    anytime({[t1, t2]}, zero, u)
```

Extensible  
library

Combinators

# Zero-Coupon Again

```
one : currency -> contract
```

```
(* if you acquire the contract (one k), then  
you acquire one unit of k. *)
```

```
scale : (observable * contract) -> contract
```

```
(* if you acquire scale(o, c), then you acquire  
c, but where all incoming and outgoing payments  
are multiplied by the value of o at acquisition  
date. *)
```

```
obs_from_float : float -> observable
```

```
(* obs_from_float k is an observable always equal to k *)
```

```
zcb(t,k,cur) =
```

```
acquire({[t]},(scale(obs_from_float k, one cur)))
```

# Summary So Far

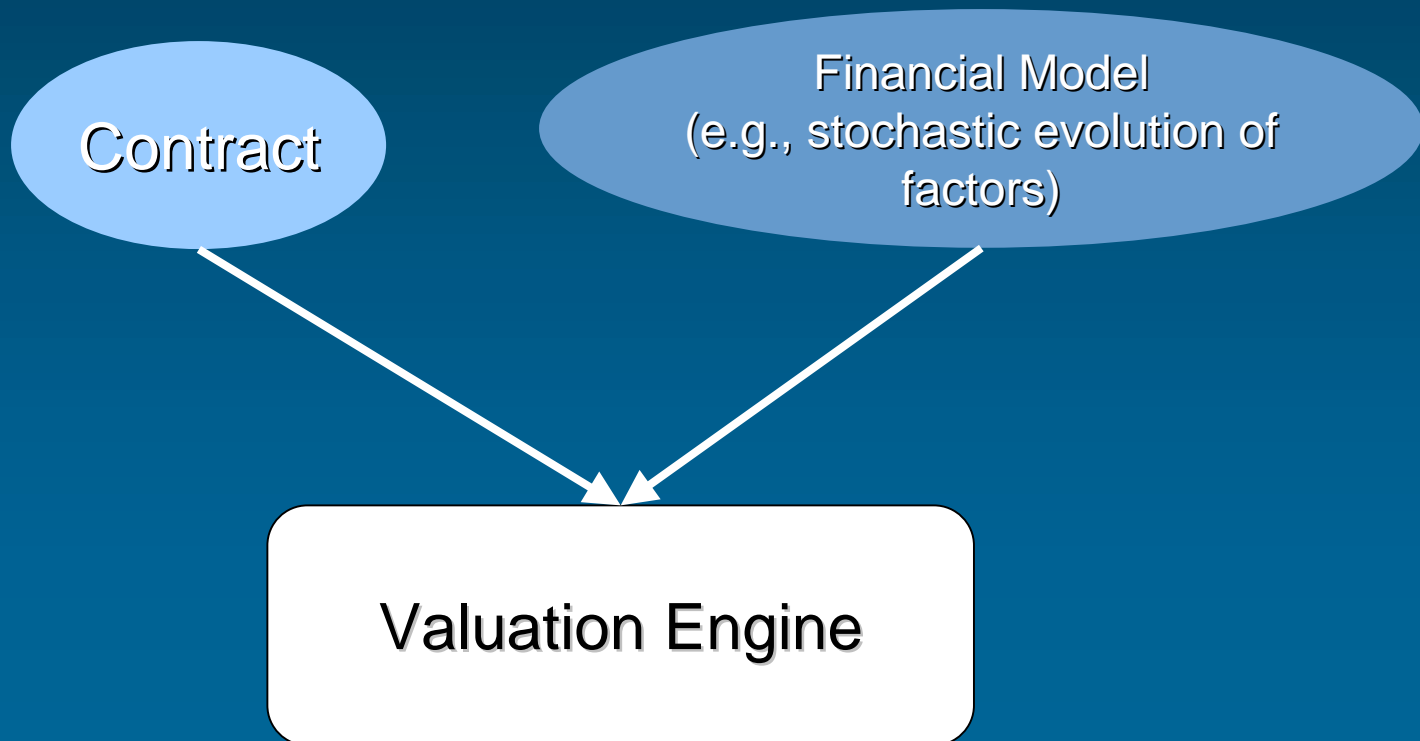
```
give      : contract -> contract
or        : (contract * contract) -> contract
and       : (contract * contract) -> contract
zero      : contract
acquire   : (region * contract) -> contract
anytime   : (region * contract * contract) -> contract
truncate  : (date * contract) -> contract
scale     : (observable * contract) -> contract
...and some more besides...
```

- ◆ *Everything* is built from the combinators!
- ◆ We need an *absolutely precise* specification of what they mean

# Valuation

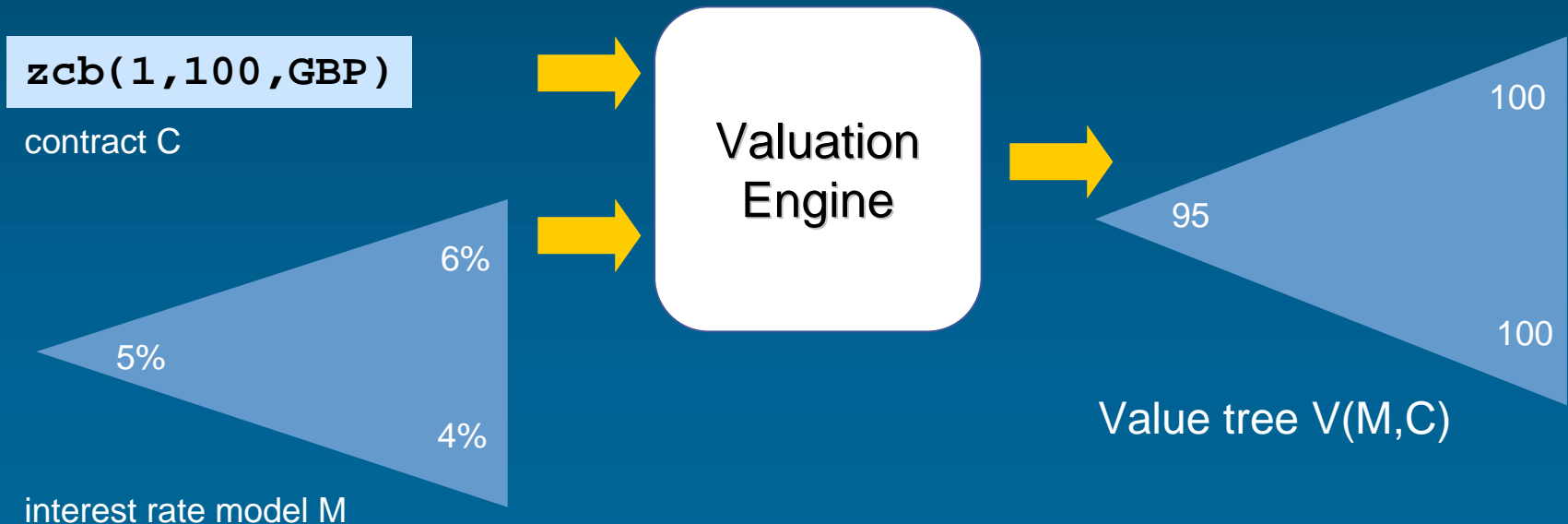
# Valuation

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



# One Possible Evaluation Model: Interest Rates "Tree"

Given a contract  $C$ , define  $V(M,C)$  to be the BDT "tree" (grid) for  $C$  under interest rate model  $M$



# Compositional Valuation

Now define  $V(M,C)$  compositionally:

$$V(M, \text{c1 and c2}) = V(M, \text{c1}) + V(M, \text{c2})$$

$$V(M, \text{c1 or c2}) = \max( V(M, \text{c1}), V(M, \text{c2}) )$$

$$V(M, \text{give c}) = - V(M, \text{c})$$

$$V(M, \text{zero}) = 0$$

...

# Compositional Valuation (2)

For the combinators *anytime* and *acquire*, we must use the abstract evaluation primitives, well known from no-arbitrage pricing theory:

$$V(M, \text{anytime}(c1, c2)) = V(M(c1)) + \text{snell}(V(M, c2) - V(M, c1))$$

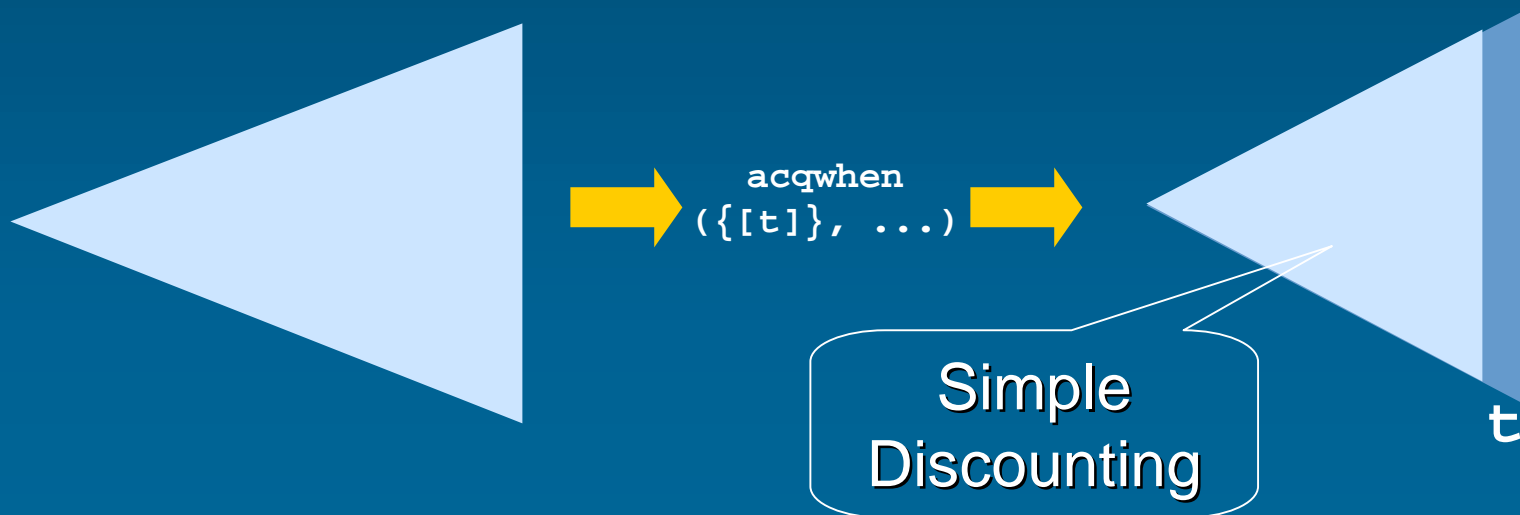
$$V(M, \text{acquire}(r, c)) = \text{discount}(r, V(M, c))$$

...

This is a major payoff! Deal with +/- 10 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



# More Precisely: What is a Model ?

- ◆ Defines its basic factors, implementation geometry
- ◆ Defines supported currencies
- ◆ Links "external" observables and contracts to "internal" pricing concepts
- ◆ Enables closed form solutions to be applied to specific (sub-) contracts
- ◆ "Accepts or refuses" to price a contract

# Intermediary Codes

Syntax check,  
error detection,  
normalisation,...

**MLFi source  
code**

Pretty-print in MLFi  
correct code

Contract level optimisation,  
dead contract elimination,  
temporal reorganisation,...

**Contract code**

**Contract code  
other state**

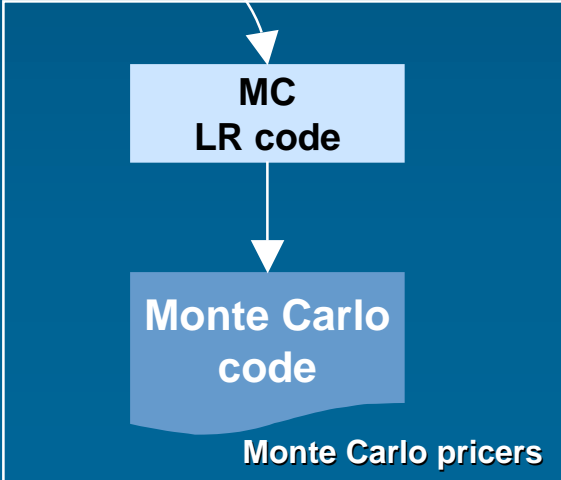
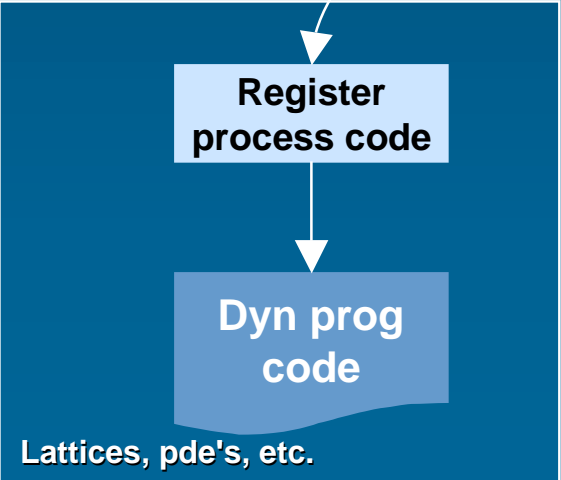
**Model def.**

Stochastic processes  
no more other types

Translate to process  
primitives + basic  
factors of model

**Process code**

Process level optimisations  
loop fusions, algebraic  
process equalities,...



# Optimizations

## ◆ Contracts

- ❖ Dead contract elimination

## ◆ Processes

- ❖ Linearity of (un)discounted evaluation
- ❖ Linearity of change of numeraire
- ❖ Temporal succession of discounting
- ❖ Evaluation sharing
- ❖ Loop fusion and closed form integration
- ❖ Loop invariant code motion

# Rescue: Compiler Technology

- ◆ Static analysis of needed calculation paths
- ◆ Data structures are computed incrementally, as they are needed (so the trees never exist in memory all at once): "slices"
- ◆ Parts that are never needed are never (or rarely) computed
- ◆ Is a deeply studied problem in theoretical computer science
- ◆ Typically: resulting code = succession of slice calculations

Slogan:

We think of the “tree” as a first class value “all at once” (semantics) but it materializes only “piecemeal” (implementation).

# Management

# Compositional Management

## Managing contract c:

- ◆ “Timely monitoring of decisions that require action (e.g., own exercise decision, etc.), events to wait for (e.g., fixing, counterparty exercise decision, etc.), amounts to pay or to receive, etc.”
- ◆ We want to derive such a “management machine” exclusively from the contract’s definition

State:

**<t, h, c, q, ls>**

Acquisition date

Manage holding up to date

Contract definition

Quantity

Long (true) or short (false)

# Compositional Management (2)

Now define a transition system compositionally:

<code>&lt;t,h,c1 and c2,q,ls&gt;</code>	<code>→ &lt;t,h,c1,q,ls&gt;</code> "and" <code>&lt;t,h,c2,q,ls&gt;</code>
<code>&lt;t,h,c1 or c2,q,true&gt;</code>	<code>→ &lt;t,h,c<sub>i</sub>,q,true&gt;</code> if I choose $c_i$
<code>&lt;t,h,c1 or c2,q,false&gt;</code>	<code>→ &lt;t,h,c<sub>i</sub>,q,false&gt;</code> if counterpart chooses $c_i$
<code>&lt;t,h,give c,q,ls&gt;</code>	<code>→ &lt;t,h,c,q,not ls&gt;</code>
<code>&lt;t,h,one k,q,ls&gt;</code>	<code>→ &lt;"empty"&gt;</code> receive ( $ls=true$ ) or pay ( $ls=false$ ) $q$ units of $k$
<code>&lt;t,h,acquire({[t1]},c),q,ls&gt;</code>	<code>→ &lt;t1,h,c,q,ls&gt;</code> when $h \geq t1$
<code>&lt;t,h,scale o c,q,ls&gt;</code>	<code>→ &lt;t,h,c,o(t)*q,ls&gt;</code>
...etc...	

Side-effect of a transition

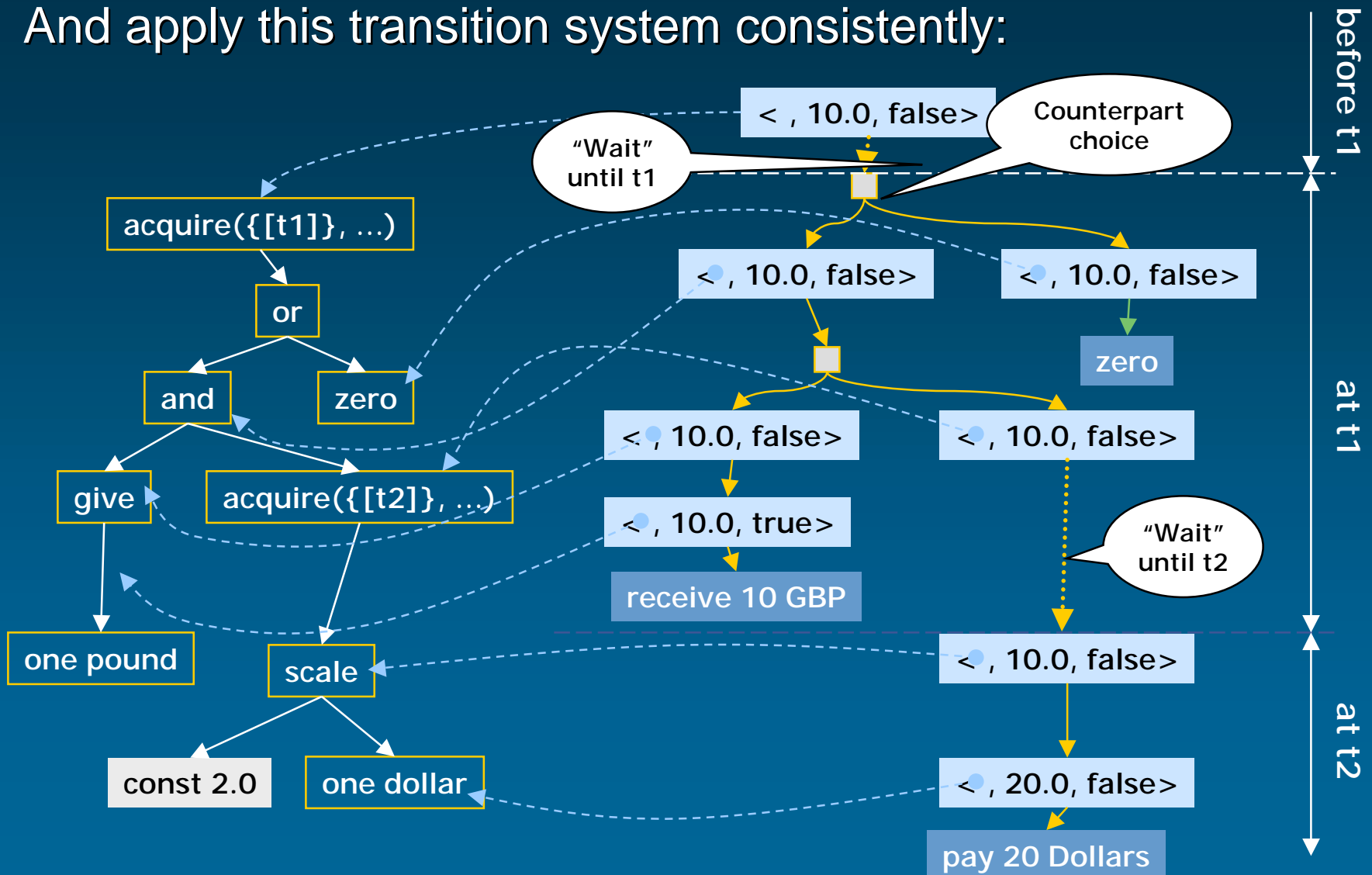
Time change

Transition condition

Decision

# Compositional Management (3)

And apply this transition system consistently:



# Management Automaton

From the *description* of a contract, we derive *formally*:

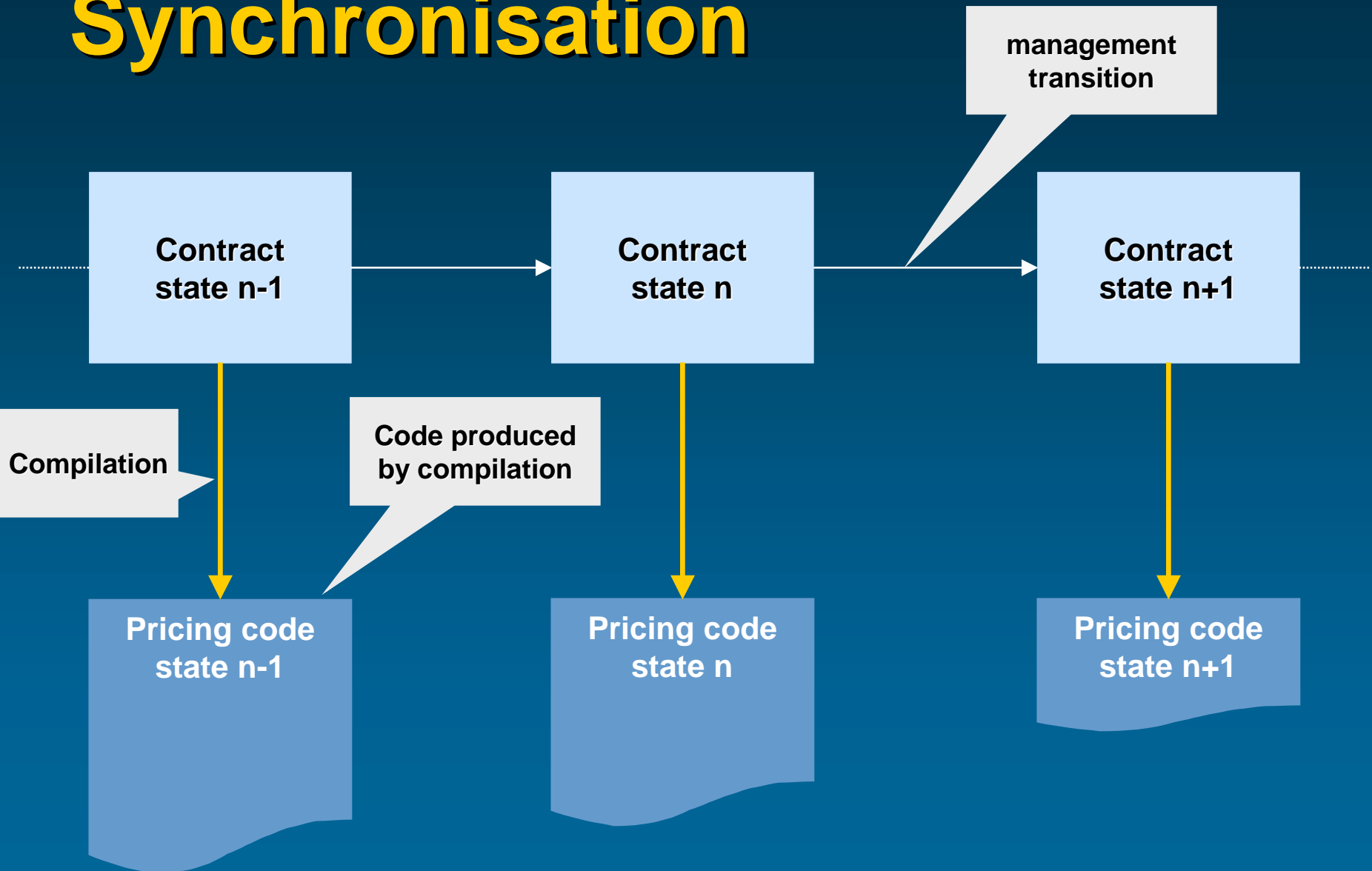
- ◆ How the contract evolves through time
- ◆ How it produces side-effects (e.g., Payments)
- ◆ How it has to wait for (and react to) external signals (e.g., Exercise decisions)

Important:

For a contract in a given state, we can always deduce easily the events (signals) we are waiting for.

We accept, of course, unexpected events (bankruptcy, legal litigation, etc.).

# Pricing and Management Synchronisation



# Events Scheduler

- ◆ The power of a combinatorial language approach can be leveraged to develop useful and innovative applications
- ◆ Example: The Contract Events Scheduler
  - ❖ Some contract events—for instance a cash payment—may depend on one or more earlier exercise decisions or fixings
  - ❖ Although dates are often known in advance, associated payments are not, as they depend on unresolved fixings
  - ❖ We want a clear, exhaustive and compact representation of the future, to be used typically by the front or the middle office

# Events Scheduler (2)

The Events Scheduler automatically populates a calendar application—Microsoft Outlook in our example—with all future events related to a given contract. No event is forgotten

- ◆ For ease of presentation, we consider an unrealistic short-dated, one-month contract
- ◆ The contract comprises a complex 3-currency option
- ◆ The same technology may be applied to any kind of contract

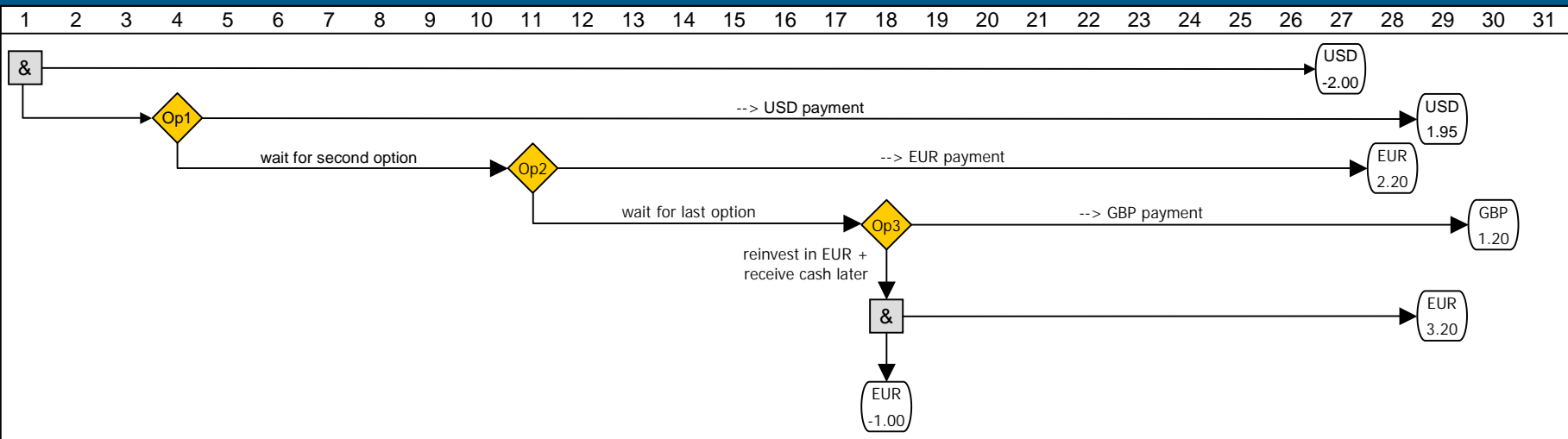
The Events Scheduler reduces operational risk

# The Financial Contract

Against the promise to pay USD 2.00 on December 27 (the price of the option), the holder has the right, on December 4, to choose between

- ❖ receiving USD 1.95 on December 29, or
- ❖ having the right, on December 11, to choose between
  - ❖ receiving EUR 2.20 on December 28, or
  - ❖ having the right, on December 18, to choose between
    - ❖ receiving GBP 1.20 on December 30, or
    - ❖ paying immediately one more EUR and receiving EUR 3.20 on December 29.

December 2001



# MLFi Contract Description

Managing and monitoring this "custom build" structure with MLFi is simple

```
let option1 =  
  
  let strike = cashflow(USD:2.00, 2001-12-27) in  
  
  let option2 =  
  
    let option3 =  
      let t = 2001-12-18T15:00 in  
      either  
        ("--> GBP payment", cashflow(GBP:1.20, 2001-12-30))  
        ("reinvest in EUR + receive cash later",  
         (give(cashflow(EUR:1.00, t))) 'and' cashflow(EUR:3.20, 2001-12-29))  
        t in  
  
    either  
      ("--> EUR payment", cashflow(EUR:2.20, 2001-12-28))  
      ("wait for last option", option3)  
      2001-12-11T15:00 in  
  
  (either  
    ("--> USD payment", cashflow(USD:1.95, 2001-12-29))  
    ("wait for second option", option2)  
    2001-12-04T15:00) 'and' (give (strike))  
  
let cal = calendar option1  
let _ = vcalfile "test.vcs" cal
```

Contract Description

Events Scheduler

# Calendar (1)

Calendar - Microsoft Outlook

File Edit View Favorites Tools Actions Help

View Go to Today Day Work Week Week Month Find Organize

Outlook Sho... Calendar December 2001

Monday	Tuesday	Wednesday	Thursday	Friday	Sat/Sun
November 26	27	28	29	30	December 1
					2
3	3:00pm Exercise: Holder ch	4	5	6	7
					8
					9
10	3:00pm Exercise<?>: Holdi	11	12	13	14
					15
					16
17	3:00pm Exercise<?>: Hold 3:00pm Payment<?>: Ho	18	19	20	21
					22
					23
24	25	26	12:00pm Payment: Holder pe	12:00pm Payment<?>: Cou	12:00pm Payment<?>: Co
					12:00pm Payment<?>: Cour
					30
					12:00pm Payment<?>: Co

My Shortcuts  
Other Shortc...  
9 Items

- ◆ All future events are posted in Outlook
- ◆ <?> denotes a contingent event

# Calendar (2)

Payment<???: Counterparty pays Holder EUR 3.20 after Exercise (Holder chose wait for second opti...

File Edit View Insert Format Tools Actions Help

Save and Close Recurrence... Invite Attendees...

Appointment Attendee Availability

Subject: Payment<???: Counterparty pays Holder EUR 3.20 after Exercise (Holder chose wait for second option), followed by Exercise (Holder chose wait for last option), followed by Exercise (Holder chose reinvest in EUR + receive cash later)

Location: This is an online meeting using: Microsoft NetMeeting

Start time: Sat 12/29/2001 12:00 PM All day event

End time: Sat 12/29/2001 12:00 PM

- ◆ <???: indicates that the payment of EUR 3.20 is contingent upon three exercise decisions
- ◆ The list of exercise decisions required for the event to occur, and precise event descriptions, are presented in chronological order:

... after Exercise (Holder chose wait for second option), followed by Exercise (Holder chose wait for last option), followed by Exercise (Holder chose reinvest in EUR + receive cash later)

# Much More to Be Said About MLFi

- ◆ The notion of *observables*
  - ❖ quoted prices (equities, bonds) and rates, etc.
  - ❖ default events, credit risk, etc.
- ◆ *Limit* options, *path-dependent* options
- ◆ MLFi's short and easy notation for *schedules* and operators over them
  - ❖ describe bonds, swaps, caps, etc.
  - ❖ but also multi-date exercisable optional structures
- ◆ Openness of LexiFi tools through generalized use of the XML standard

# Summary

# 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 (front office), management (back office) and other purposes
- ◆ A suitable formalism: *reasoning about contracts, proofs about contracts, simplifications of contracts, transforming contracts*

# Summary (2)

- ◆ LexiFi develops and markets compiler technology and open applications built around the MLFi language, using the XML standard for easy interoperability
- ◆ For more information:

[www.lexifi.com](http://www.lexifi.com)