Composing Contracts

This document is an uComposing Contracts

This document is an unofficial example implementation of the system originally described in the paper Composing contracts: an adventure in financial engineering, by Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Familiarity with both versions of this paper is assumed.
This example implementation is a literate Haskell program, Contracts.lhs. Don't be put off by the embedded HTML — it is an executable Haskell program. It can be executed directly and experimented with in an interactive Haskell environment such as GHCI.
The program is completely self contained, depending only on a few GHC libraries. Compatibility with other Haskell implementations has not been tested.
A web interface to some of the examples is also available.
Please note that this program is an example, intended only for educational use in its current form. The core implementation is only 215 lines of code, excluding examples and user interface. As such, it has many limitations. See the Future work section for further information.

Contents

A note about the original papers

There are a few significant differences between the first and second papers. The change with the most impact on the implementation relates to the representation of contract horizons.
In A:5.3, entitled "Implementation in Haskell", a value process representation is described in which the lattice for a value process is stored as a list of random variables "in reverse time order", i.e. horizon first, along with the horizon's timestep number. This is possible because the first paper uses a simple, definite approach to representing horizons: they are specified by a single date.
However, this implementation is not compatible with some of the design decisions described in the second paper (B). Since the horizon of a contract can depend on observables other than the date, the second paper introduces a more sophisticated approach to representing horizons: horizons are specified by boolean value processes (type PR Bool) that define contract acquisition regions.

This allows the horizon of a contract to cross over more than one time step, depending on the value of the observable(s) that define the horizon. The use of boolean value processes "to describe the 'region' in which one can acquire a contract" is described as a "breakthrough" in section 3.6 of the second paper.
Since contracts can have indefinite horizons, and also because some value processes may have no horizon, it's not possible in general to represent a value process horizon-first, as suggested in the first paper.
Since the second paper does not provide an explicit description of the Haskell representation, our implementation uses a variation on the one described in the first paper, with some changes to support the second paper's design.

Module header

```
module Contracts
 -- (renderEx, renderExDefault, ExContr(ExContr)) -- limit exports for use from HAppS
 where

import List
 import Numeric
 import Control.Monad
 import System
 import Text.XHtml.Strict
 import Data.Unique
```

Notational conventions

Most references in this document are of the form P:S.s, where e.g. A:3.5 refers to paper A, section 3.5. The papers are listed in the References section. Papers A and B are the first and second versions of the Composing Contracts paper, and paper C is a paper about the functional reactive programming system Fran.
Notational conventions from B:Fig.1:

```
c, d, u : Contract
      o : Observable
   t, s : Date, time
      k : Currency
      x : Dimensionless real value
      p : Value process
      v : Random variable
```

Basic data types

```
data Currency = USD | GBP | EUR | ZAR | KYD | CHF  deriving (Eq, Show)
```

A Date is represented as a pair consisting of the start date/time of a contract, represented by a value of type CalendarTime, and an integer representing the number of time steps since the start of a contract. For example purposes, the time steps are of unspecified duration, and the CalendarTime type is stubbed out.
Representing the time step as a separate integer is useful when manipulating trees representing the evolution of a process over time, where the time step corresponds to an index into a list.

```
type Date = (CalendarTime, TimeStep)

type TimeStep = Int
 type CalendarTime = ()
```

Since the example doesn't use real dates, mkDate cheats and creates a Date from a TimeStep.

```
mkDate :: TimeStep -> Date
 mkDate s = ((),s)
```

Because real dates aren't used, all value processes are assumed to begin at the same time, the zeroth time step, time0.

```
time0 :: Date
 time0 = mkDate 0
```

This simplifies some aspects of the implementation, discussed further under Support actual dates and times.

Contract implementation

The representation of a contract is based on A:5.3.

```
data Contract =
    Zero
  | One  Currency
  | Give Contract
  | And  Contract Contract
  | Or   Contract Contract
  | Cond    (Obs Bool)   Contract Contract
  | Scale   (Obs Double) Contract
  | When    (Obs Bool)   Contract
  | Anytime (Obs Bool)   Contract
  | Until   (Obs Bool)   Contract
  deriving Show
```

Observable data type

"In general, a value of type Obs *d* represents a time-varying quantity of type *d*." (A:3.3)
An obvious implementation might involve a function of type (Date -> a). However, a "quantity" in this context is not a single value, but rather a random variable, i.e. a set of possible values. This suggests a function of type (Date -> RV a). This would allow an arbitrary observable to be converted to a value process during evaluation of a contract, by applying successive dates to the observable's function. This would be a valid implementation, but in order to take maximum advantage of the lazy list representation used for value processes, we will actually use the following type:

```
newtype Obs a = Obs (Date -> PR a)
```

An observable is thus represented as a function from a starting date to a value process. The "time-varying" nature of an observable is captured primarily by the value process itself (PR a); the Date in the function's type is simply used to specify the start date for the resulting value process.
For development and debugging purposes, Obs will be showable. Since it is a function type, this is achieved by applying it to a dummy date and displaying the first slice of the resulting process. This is only useful for getting a rough idea of the definition of a contract. See Enhance observable representation for further discussion of this.

```
instance Show a => Show (Obs a) where
  show (Obs o) = let (PR (rv:_)) = o time0 in "(Obs " ++ show rv ++ ")"
```

Primitives for Defining Contracts

Define a combinator interface to the Contract datatype. From B:Fig.2:

```
zero :: Contract
 zero = Zero

one :: Currency -> Contract
 one = One

give :: Contract -> Contract
 give = Give

cAnd :: Contract -> Contract -> Contract
 cAnd = And

cOr :: Contract -> Contract -> Contract
 cOr = Or
```

```
cond :: Obs Bool -> Contract -> Contract -> Contract
 cond = Cond

scale :: Obs Double -> Contract -> Contract
 scale = Scale

cWhen :: Obs Bool -> Contract -> Contract
 cWhen = When

anytime :: Obs Bool -> Contract -> Contract
 anytime = Anytime

cUntil :: Obs Bool -> Contract -> Contract
 cUntil = Until
```

Derived combinators

Other combinators can now be derived from these primitives, e.g.:

```
andGive :: Contract -> Contract -> Contract
 andGive c d = c `cAnd` give d
```

Primitives over observables

konst x is an observable that has value x at any time.

```
konst :: a -> Obs a
 konst k = Obs (\t -> bigK k)
```

lift f o is the observable whose value is the result of applying f to the value of the observable o.

```
lift :: (a -> b) -> Obs a -> Obs b
 lift f (Obs o) = Obs (\t -> PR $ map (map f) (unPr $ o t))
```

$lift_2$ o_1 o_2 is the observable whose value is the result of applying f to the values of the observables o_1 o_2.

```
lift2 :: (a -> b -> c) -> Obs a -> Obs b -> Obs c
 lift2 f (Obs o1) (Obs o2) = Obs (\t -> PR $ zipWith (zipWith f) (unPr $ o1 t) (unPr $ o2 t))
```

"The value of the observable date at date t is just t."

```
date :: Obs Date
 date = Obs (\t -> PR $ timeSlices [t])
```

"All numeric operations lift to the Obs type. The implementation is simple, using lift and $lift_2$."

```
instance Num a => Num (Obs a) where
  fromInteger i = konst (fromInteger i)
  (+) = lift2 (+)
  (-) = lift2 (-)
  (*) = lift2 (*)
  abs = lift abs
  signum = lift signum
```

One quirk is that we need to define a stub for Eq to support the Num instance.

```
instance Eq a => Eq (Obs a) where
  (==) = undefined
```

We can't implement Eq on an Observable's function, but we can provide a lifted version of equality:

```
(==*) :: Ord a => Obs a -> Obs a -> Obs Bool
 (==*) = lift2 (==)
```

at is a boolean observable that becomes True at time t (B:3.2)

```
at :: Date -> Obs Bool
 at t = date ==* (konst t)
```

Typeclasses don't work so well for relational operators, so define a separate family of them (B:3.3)

```
(%<), (%<=), (%=), (%>=), (%>) :: Ord a => Obs a -> Obs a -> Obs Bool
 (%<)  = lift2 (<)
 (%<=) = lift2 (<=)
 (%=)  = lift2 (==)
 (%>=) = lift2 (>=)
 (%>)  = lift2 (>)
```

Option contracts

From B:3.4:

```
european :: Date -> Contract -> Contract
 european t u = cWhen (at t) (u `cOr` zero)

american :: (Date, Date) -> Contract -> Contract
 american (t1, t2) u = anytime (between t1 t2) u
```

```
between :: Date -> Date -> Obs Bool
 between t1 t2 = lift2 (&&) (date %>= (konst t1)) (date %<= (konst t2))
```

Value processes

A value process PR a is represented as a list of random variables RV a, with the random variable corresponding to the earliest time step appearing first in the list.

```
newtype PR a = PR { unPr :: [RV a] } deriving Show
```

Note that the "informal type definition" of a value process is given in B:4.1 as PR a = Date -> RV a. However, this definition should not be taken literally. Among other things, it is not amenable to efficient list-based recursive processing of entire value processes, since it requires a lookup for access to each successive date. (This was discovered the hard way in an earlier implementation of this code — thanks to Chung-chieh Shan for pointing out the advantages of relying pervasively on a lazy list implementation, during the presentation of the earlier version of this code in NYC.)
A random variable RV a describes the possible values for a value process at a particular time step. For example, the random variable describing the outcome of a dice throw would be[1,2,3,4,5,6]. Random variables are therefore implemented as simple lists.

```
type RV a = [a]
```

Value process helpers

takePr truncates a (possibly infinite) value process.

```
takePr :: Int -> PR a -> PR a
 takePr n (PR rvs) = PR $ take n rvs
```

horizonPr determines the number of time steps in a value process. Only terminates for finite value processes.

```
horizonPr :: PR a -> Int
 horizonPr (PR rvs) = length rvs
```

andPr returns True if every value in a value process is true, false otherwise. Only terminates for finite value processes.

```
andPr :: PR Bool -> Bool
 andPr (PR rvs) = and (map and rvs)
```

Model

The model specifies the particular semantics for underlying observables such as the evolution of interest rates, exchange rates, and the types of calculation used. The contract evaluation function, evalC, is parameterized over a model to allow different models to be easily used. The model itself is implemented as a record of model-specific data and functions which can easily be instantiated by a function such as exampleModel below. Essentially, this amounts to a poor man's higher-order module.

```
data Model = Model {
  modelStart :: Date,
  disc       :: Currency -> (PR Bool, PR Double) -> PR Double,
  exch       :: Currency -> Currency -> PR Double,
  absorb     :: Currency -> (PR Bool, PR Double) -> PR Double,
  rateModel  :: Currency -> PR Double
  }
```

Define a specific model which defines the model functions given in the paper. This would normally be defined in a separate module.

```
exampleModel :: CalendarTime -> Model
 exampleModel modelDate = Model {
  modelStart = (modelDate,0),
  disc       = disc,
  exch       = exch,
  absorb     = absorb,
  rateModel  = rateModel
  }

 where
```

The example model's functions are defined in the following sections. Note that these definitions are local to the exampleModel record definition (due to the where clause above).

Interest rate model

See B:5.1. This constructs a lattice containing possible interest rates given a starting rate and an increment per time step. This "unrealistically regular" model matches that shown in B:Fig.8. However, it is so simple that some interest rates go negative after a small number of time steps. A better model is needed for real applications. Don't use this to model your retirement fund!

```
 rates :: Double -> Double -> PR Double
  rates rateNow delta = PR $ makeRateSlices rateNow 1
    where
      makeRateSlices rateNow n = (rateSlice rateNow n) : (makeRateSlices (rateNow-delta)
(n+1))
```

```
    rateSlice minRate n = take n [minRate, minRate+(delta*2) ..]
```

Each currency has different parameters for the interest rate model. Since the model is not realistic, these parameters are currently entirely arbitrary.

```
 rateModels = [(CHF, rates 7   0.8)
              ,(EUR, rates 6.5 0.25)
              ,(GBP, rates 8   0.5)
              ,(KYD, rates 11  1.2)
              ,(USD, rates 5   1)
              ,(ZAR, rates 15  1.5)
              ]

 rateModel k =
    case lookup k rateModels of
      Just x -> x
      Nothing -> error $ "rateModel: currency not found " ++ (show k)
```

'Disc' primitive

The primitive (disc t k) maps a real-valued random variable at date T, expressed in currency k, to its "fair" equivalent stochastic value process in the same currency k. See B:4.4 and B:Fig.7.
A simplifying assumption is that at some point, the boolean-valued process becomes True for an entire RV. This provides a simple termination condition for the discounting process.

```
 disc :: Currency -> (PR Bool, PR Double) -> PR Double
  disc k (PR bs, PR rs) = PR $ discCalc bs rs (unPr $ rateModel k)

   where

     discCalc :: [RV Bool] -> [RV Double] -> [RV Double] -> [RV Double]
      discCalc (bRv:bs) (pRv:ps) (rateRv:rs) =
        if and bRv -- test for horizon
          then [pRv]
          else let rest@(nextSlice:_) = discCalc bs ps rs
                   discSlice = zipWith (\x r -> x / (1 + r/100)) (prevSlice nextSlice) rateRv
                   thisSlice = zipWith3 (\b p q -> if b then p else q) -- allow for partially discounted
slices
                                 bRv pRv discSlice
              in thisSlice : rest
```

prevSlice calculates a previous slice in a lattice by averaging each adjacent pair of values in the specified slice

```
    prevSlice :: RV Double -> RV Double
     prevSlice [] = []
     prevSlice (_:[]) = []
     prevSlice (n1:rest@(n2:_)) = (n1+n2)/2 : prevSlice rest
```

'Absorb' primitive

"Given a boolean-valued process o, the primitive absorb$_k$(o,p) transforms the real-valued process p, expressed in currency k, into another real-valued process. For any state, the result is the expected value of receiving p's value if the region o will never be True, and receiving zero in the contrary. In states where o is True, the result is therefore zero."

```
  absorb :: Currency -> (PR Bool, PR Double) -> PR Double
   absorb k (PR bSlices, PR rvs) =
     PR $ zipWith (zipWith $ \o p -> if o then 0 else p)
                 bSlices rvs
```

Exchange rate model

This is a stub which always returns 1.

```
  exch :: Currency -> Currency -> PR Double
   exch k1 k2 = PR (konstSlices 1)
```

The definition of the exampleModel ends here.

Expected value

The code for absorb above does not obviously deal with the expected value mentioned in the spec. This is because the expected value of each random variable is implicit in the value process lattice representation: each node in the lattice is associated with a probability, and the expected value at a particular date is simply the sum of the product of the value at each node and its associated probability. The following functions implement this calculation.

```
expectedValue :: RV Double -> RV Double -> Double
 expectedValue outcomes probabilities = sum $ zipWith (*) outcomes probabilities

expectedValuePr :: PR Double -> [Double]
 expectedValuePr (PR rvs) = zipWith expectedValue rvs probabilityLattice
```

Snell primitive

Not currently implemented. The paper describes the following as a possible algorithm:

- take the final column of the tree (horizon),
- discount it back one time step,

- take the maximum of that column with the corresponding column of the original tree,
- then repeat that process all the way back to the root.

$snell_k(o,p)$ is the smallest process q (under an ordering relation mention briefly at the end of B:4.6) such that:
forall o' . (o => o') => q >= $snell_k(o',q)$
That is, an American option is the least upper bound of any of the deterministic acquisition choices specified by o', where o' is a sub-region of o.

Probability calculation

Each node in a value process lattice is associated with a probability.
"...in our very simple setting the number of paths from the root to the node is proportional to the probability that the variable will take that value."

```
probabilityLattice :: [RV Double]
 probabilityLattice = probabilities pathCounts
   where

    probabilities :: [RV Integer] -> [RV Double]
     probabilities (sl:sls) = map (\n -> (fromInteger n) / (fromInteger (sum sl))) sl : probabilities
sls
```

To calculate the number of paths to each node in a lattice, simply add the number of paths to the pair of parent nodes. This needs to work with Integers as opposed to Ints, because:

```
findIndex (\sl -> maximum sl > (fromIntegral (maxBound::Int))) pathCounts ==> Just 67
    pathCounts :: [RV Integer]
     pathCounts = paths [1] where paths sl = sl : (paths (zipWith (+) (sl++[0]) (0:sl)))
```

Compositional valuation semantics for contracts

See B:Fig.4. A Haskell type signature for eval is specified in A:5.3. It has been modified here to return a PR Double, as specified in the semantics in Figure 4, instead of a ValProc. (In this implementation, the PR Double type is essentially equivalent to the first paper's ValProc type.)

```
evalC :: Model -> Currency -> Contract -> PR Double
 evalC (Model modelDate disc exch absorb rateModel) k = eval    -- punning on record
fieldnames for conciseness
   where eval Zero           = bigK 0
         eval (One k2)       = exch k k2
         eval (Give c)       = -(eval c)
         eval (o `Scale` c)  = (evalO o) * (eval c)
         eval (c1 `And` c2)  = (eval c1) + (eval c2)
         eval (c1 `Or` c2)   = max (eval c1) (eval c2)
         eval (Cond o c1 c2) = condPr (evalO o) (eval c1) (eval c2)
```

```
    eval (When o c)     = disc   k (evalO o, eval c)
--    eval (Anytime o c)  = snell  k (evalO o, eval c)
    eval (Until o c)    = absorb k (evalO o, eval c)
```

Valuation semantics for observables

See B:Fig.5. The evalation function for observables, evalO, converts an observable's function to a value process by applying the function to a start date.

```
evalO :: Obs a -> PR a
 evalO (Obs o) = o time0
```

Process primitives

B:Fig6

```
bigK :: a -> PR a
 bigK x = PR (konstSlices x)

konstSlices x = nextSlice [x]
   where nextSlice sl = sl : (nextSlice (x:sl))

datePr :: PR Date
 datePr = PR $ timeSlices [time0]

timeSlices sl@((s,t):_) = sl : timeSlices [(s,t+1) | _ <- [0..t+1]]

condPr :: PR Bool -> PR a -> PR a -> PR a
 condPr = lift3Pr (\b tru fal -> if b then tru else fal)

liftPr :: (a -> b) -> PR a -> PR b
 liftPr f (PR a) = PR $ map (map f) a

lift2Pr :: (a -> b -> c) -> PR a -> PR b -> PR c
 lift2Pr f (PR a) (PR b) = PR $ zipWith (zipWith f) a b

lift2PrAll :: (a -> a -> a) -> PR a -> PR a -> PR a
 lift2PrAll f (PR a) (PR b) = PR $ zipWithAll (zipWith f) a b

lift3Pr :: (a -> b -> c -> d) -> PR a -> PR b -> PR c -> PR d
 lift3Pr f (PR a) (PR b) (PR c) = PR $ zipWith3 (zipWith3 f) a b c
```

A version of zipWith that handles input lists of different lengths. This is used to support lifted binary operations such as (+).

```
zipWithAll :: (a -> a -> a) -> [a] -> [a] -> [a]
 zipWithAll f (a:as) (b:bs)     = f a b : zipWithAll f as bs
 zipWithAll f as@(_:_) []       = as
 zipWithAll f []      bs@(_:_) = bs
 zipWithAll _ _       _        = []
```

```
instance Num a => Num (PR a) where
   fromInteger i = bigK (fromInteger i)
   (+) = lift2PrAll (+)
   (-) = lift2PrAll (-)
   (*) = lift2PrAll (*)
   abs = liftPr  abs
   signum = liftPr signum
```

```
instance Ord a => Ord (PR a) where
   max = lift2Pr max
```

```
instance Eq a => Eq (PR a) where
   (PR a) == (PR b) = a == b
```

Examples

Instantiate the example model with what would be the model's starting date, if real dates were used.

```
xm :: Model
 xm = exampleModel ()
```

Define an evaluator specific to the example model and the USD currency.

```
evalX :: Contract -> PR Double
 evalX = evalC xm USD
```

The by-now-infamous zero-coupon bond:

```
zcb :: Date -> Double -> Currency -> Contract
 zcb t x k = cWhen (at t) (scale (konst x) (one k))
```

A contract using the ZCB:

```
c1 :: Contract
 c1 = zcb t1 10 USD
```

The test date for the bond is horizon timesteps from the model's current date:

```
t1 :: Date
 t1 = mkDate t1Horizon

t1Horizon = 3 :: TimeStep
```

A stripped-down versions of the European option from B:3.4. That example uses real dates that range over more than two years. This is a smaller version. Its results have not been checked.

```
c11 :: Contract
 c11 = european (mkDate 2)
         (zcb (mkDate 20) 0.4 USD `cAnd`
          zcb (mkDate 30) 9.3 USD `cAnd`
          zcb (mkDate 40) 109.3 USD `cAnd`
          give (zcb (mkDate 12) 100 USD))
```

Evaluate the contract c1, in dollars, to produce a value process:

```
pr1 :: PR Double
 pr1 = evalX c1
```

Access the underlying lattice (list of slices):

```
tr1 = unPr pr1
```

Test of 'cUntil' - implementation of absorbEx is similar to zcb, but uses cUntil instead of cWhen.

```
absorbEx t x k = cUntil (konst t %> date) (scale (konst x) (one k))
```

Main function

There is no main function. This program is intended to be run in an interactive Haskell environment such as GHCI, where the above examples, combinators, and evaluation functions can be examined and experimented with.
A web interface to some of the examples is also available.

Pretty pictures

Process lattice for zcb

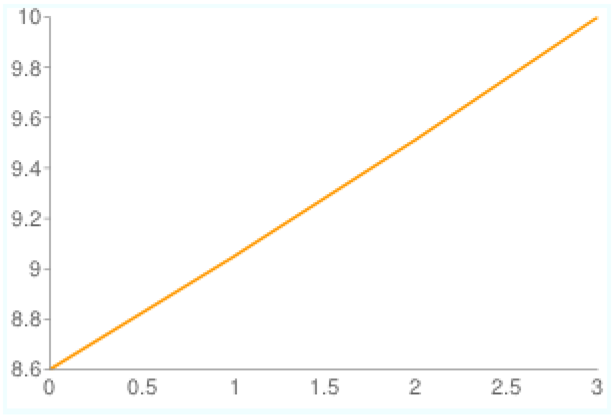
The following diagram shows the value process lattice for the contract (zcb (mkDate 3) 10 USD). It matches B:Fig.9, except for minor details such as the fetching shade of pink. It was generated using GraphViz, by the following code.

```
zcbImage = latticeImage pr1 "fig9" "png"
```

Expected value

The following is a chart of expected value at each timestep. The implementation uses the Google Chart API.

```
c1ExpectedValueUrl = chartUrl $ expectedValuePr pr1
```



Interest rate evolution

This is the short-term interest rate evolution from B:Fig8.

```
rateEvolution = latticeImage (takePr (t1Horizon + 1) $ rateModel xm USD) "fig8" "png"
```

Future work

In its present state, this implementation is very much a prototype, intended mainly to provide a concrete illustration of concepts described in the papers on which it is based. The core implementation is less than 200 lines of Haskell code, excluding examples and user interface. As such, it has many limitations and omissions. Some of the more obvious enhancements that could be made include:

- Support actual dates and times.
- Implement monadic contract evaluator.
- Enhance observable representation.
- Improve contract horizon handling.
- Implement more realistic interest and exchange rate models.
- Implement the snell function.

- Implement a model using other kinds of numerical methods, such as Monte Carlo.
- Implement models for specific observables.
- Factoring into modules.

Additional detail for selected items follows.

Support actual dates and times

As described in Basic data types, time is currently modeled using abstract integer time steps. Adding basic support for real dates and times should not be difficult. Since many financial contracts do not need to be concerned with time steps smaller than days, the following description will focus only on date handling, but the same design applies to support for real times.

Adding date support requires changes in the following areas:

- Value processes should store their start date in order to correctly handle operations involving processes that start at different dates.
- Operations on value processes need to check the start dates of the processes they operate on, and proceed accordingly. For example, when adding two value processes that begin at different dates, only random variables which represent the same time step should be added together.
- Functions that rely on the zeroth time step time0, particularly the observable evalution function evalO and the date process datePr, should instead use the start date of the contract evaluator'sModel parameter. This can be achieved by converting the contract evaluator evalC from direct style to monadic style (see next subsection), which will allow the functions in question access to the model's start date.

Implement monadic contract evaluator

The evaluator for contracts is currently a very simple, direct-style implementation. This is possible in part because of simplifying choices such as the use of time steps without actual dates, as mentioned above.

Many other kinds of enhancements to the implementation are likely to require a more sophisticated evaluator design. Converting to a monadic evaluator would support such enhancements. Aside from providing primitives direct access to the model, it would also allow alternate monads and monad transformers to be used to parameterize the evaluator semantics. Jeff Polakow pointed out that a probability monad could be useful, for example.

Enhance observable representation

Two ways in which the representation of observables might be enhanced are as follows:

- C:4.1 suggests that the type could be designed to support simplification and other changes to the observable over time, as well as efficient handling of time intervals (C:4.2). The details here depend on the requirements of the system being developed.
- As the definition of the Show instance for the Observable data type demonstrates, the representation of observables as functions limits the ability to inspect contract definitions. With an embedded DSL, if metadata is not stored along with the DSL

terms, then the host language source code may be the only complete specification of embedded terms that involve functions. This could be addressed by the use of a tag to identify observables. This might take the form of a sum type representing primitive observables such as constants and dates, with provision for arbitrary named functions for more complex observables. Such a representation is hinted at in the description of the valuation semantics for observables in B:Fig.5.

Improve contract horizon handling

The second paper's use of boolean value processes to represent acquisition regions is very general. The full generality of this model is not exploited by the current implementation. For example, thedisc primitive assumes that the horizon of a contract corresponds to a single random variable. However, contract horizons may be based on more complex observables than the date, and composed contracts may also result in complex scenarios in which a contract's horizon crosses more than one random variable.
In addition, some value processes are infinite and have no horizons. The system should track this to allow it to prevent attempts to perform non-terminating operations on such contracts. This may also help in implementing operations that combine contracts with different horizons.

Appendix A - Tests

A few small tests that came up during development.

```
tolerance = 0.001
```

Test of constant process:

```
testK = andPr $ liftPr (== 100) $ takePr 10 (bigK 100)
```

Test that a slice in the probability lattice adds up to probability 1.0:

```
testProb = (sum $ probabilityLattice !! 100) - 1 < tolerance
```

Test the result of evaluating the c1 contract

```
testPr1 = andPr $ lift2Pr (\a b -> (abs (a - b)) < tolerance)
                         pr1
                         (PR [[8.641], [9.246,8.901], [9.709,9.524,9.346], [10,10,10,10]])
```

Run all tests (all three of them!)

```
tests = and [testK
            ,testProb
            ,testPr1]
```

Appendix B - HTML table output for value process

This renders a value process lattice as a kind of pyramid, using an HTML table.

```
prToTable pr@(PR rvs) = table << (snd $ foldl renderSlice (0, noHtml) rvs)
  where

   horizon = horizonPr pr
    renderSlice (n, rows) rv = (n+1, rows +++ (tr $ td << (show n)
                                            +++ (spacer $ horizon - n)
                                            +++ (concatHtml (map renderCell rv))
                                            +++ (spacer $ horizon - n + 1)))

   renderCell v = td ! [theclass "cell", colspan 2] << (showFFloat (Just 2) v "")

spacer 0 = noHtml
 spacer n = td ! [theclass "sp", colspan n] << noHtml
```

Appendix C - Graphviz Output

This code generates graphs which represent a value process lattice. Currently assumes Double values, constrained by showNode's formatting of the value.
Write out tree as Dot file and run Dot to generate image:

```
latticeImage :: PR Double -> String -> String -> IO ExitCode
 latticeImage pr baseName imageType =
  do writeTreeAsDot baseName pr
     runDot baseName imageType
```

Supports interactive display of generated Dot code.

```
printTree :: PR Double -> IO ()
 printTree pr = mapM_ putStrLn (dotGraph (prToDot pr))
```

Write a value process out as a Dot file.

```
writeTreeAsDot :: String -> PR Double -> IO ()
 writeTreeAsDot baseName pr = writeFile (baseName ++ dotExt) $ unlines (dotGraph
(prToDot pr))
```

Run Dot on a file with the specified base name, and generate a graphic file with the specified type.

```
runDot :: String -> String -> IO ExitCode
 runDot baseName fileType =
  system $ concat ["dot -T", fileType,
                   " -o ", baseName, ".", fileType, " ",
                   baseName, dotExt]
```

Convert a (PR Double) to a list of dot node relationships.

```
prToDot :: PR Double -> [String]
 prToDot (PR rvs) = rvsToDot rvs
```

Convert lattice to list of dot node relationships.

```
rvsToDot :: [RV Double] -> [String]
 rvsToDot rvs = let numberedRvs = assignIds rvs 1
                in showNodes numberedRvs ++ treeToDot numberedRvs

dotExt = ".dot"
```

Number each of the nodes in a lattice.

```
assignIds :: [RV a] -> Int -> [RV (Int, a)]
 assignIds [] n = []
 assignIds (rv:rvs) n = numberList (reverse rv) n : assignIds rvs (n + length rv)

numberList :: [a] -> Int -> [(Int, a)]
 numberList l n = zip [n .. n + length l] l
```

showNodes returns a list of "primary" Dot representations of numbered RV nodes, with each node's value specified as the node's label. These nodes can then be referenced repeatedly in the generated Dot code without specifying a label.

```
showNodes :: [RV (Int, Double)] -> [String]
 showNodes numberedRvs = concatMap showSlice (numberList numberedRvs 0)
   where showSlice (n, sl) = ("subgraph Slice" ++ show n ++ " { rank=same")
                          : (map (\(n,s) -> show n ++ nodeLabel s) sl)
                          ++ ["SL" ++ (show n) ++ " [label=\"" ++ show n ++ "\" style=solid
peripheries=0] }"]

nodeLabel :: Double -> String
 nodeLabel s = " [label=\"" ++ (showFFloat (Just 2) s "\"]")
```

generate Dot code for relationships between numbered RV nodes.

```
treeToDot :: [RV (Int, a)] -> [String]
 treeToDot [a] = []
 treeToDot (a:b:rest) = dotJoin a (take (length a) b)
                     ++ dotJoin a (tail b)
                     ++ treeToDot (b:rest)

dotJoin :: RV (Int, a) -> RV (Int, a) -> [String]
 dotJoin a b = zipWith (\(m,a) (n,b) -> (show m) ++ " -- " ++ (show n)) a b
```

```
dotGraph :: [String] -> [String]
 dotGraph body = dotGraphHdr ++ (map formatDotStmt body) ++ ["}"]

dotGraphHdr :: [String]
 dotGraphHdr = ["graph contract_lattice {"
               ,"  rankdir=LR;"
               ,"  dir=none;"
               ,"  node [style=filled color=pink shape=box fontsize=10 width=0.5 height=0.4];"]

formatDotStmt :: String -> String
 formatDotStmt s = "  " ++ s ++ ";"
```

Appendix D - Google chart

This generates a URL for the Google Chart API. Used for expected value chart.

```
chartUrl :: [Double] -> String
 chartUrl vs =
"http://chart.apis.google.com/chart?chs=300x200&cht=lc&chxt=x,y&chg=20,25,2,5&chxr=0,0,"
             ++ (show $ length vs - 1)
             ++ "|1," ++ (showFFloat (Just 1) ymin ",")
                     ++ (showFFloat (Just 1) ymax "&chd=t:")
             ++ (concat $ intersperse "," $ map (\y -> showFFloat (Just 1) y "") ys)
  where (ymin, ymax, ys) = chartScale vs 100
```

Scale specified list of values to a range between 0 and upper.

```
chartScale ys upper =
  let ymin = minimum ys
      ymax = maximum ys
      yrange = ymax - ymin
      yscale = upper/yrange
  in (ymin, ymax, map (\y -> (y - ymin) * yscale ) ys)
```

Appendix E - Web interface

The following code implements a very simple web interface, which allows a few canned examples to be run and displays the resulting value process lattice images along with a chart of expected value (where appropriate).
At the time of writing, the web interface is running at http://contracts.scheming.org/contractEx. This URL may change in future.

Serializable example specification

ExContr is a type to specify examples to be run, which is serialized in the request URL.

```
newtype ExContr = ExContr (String, [Double], Bool) deriving (Read,Show,Eq)

useLatticeImage (ExContr (_, _, b)) = b

 webPath = "/home/anton/usr/happs.org/public/"
 -- webPath = "/home/anton/happs91/HAppS-Begin/public/"

 tmpImgPath = "imgtmp/"

 baseDotFilename = "pr-lattice"

 pageTitle = "Composing contracts - simple charts"

mkUniqueName :: String -> IO String
 mkUniqueName baseName =
   do u <- newUnique
      return $ baseName ++ (show $ hashUnique u)

renderEx :: ExContr -> IO Html
 renderEx exSpec@(ExContr (contractId, args, lattice)) =
   let pr = evalEx exSpec
       expValChart = if contractId == "probs" then noHtml -- expected value is meaningless for
the probabilities it relies on
                     else h3 << "Expected value" +++ image ! [src (chartUrl $ expectedValuePr pr)]
       imageType = "png"
   in if useLatticeImage exSpec
      then do baseName <- mkUniqueName baseDotFilename
              exitCode <- latticeImage pr (webPath ++ tmpImgPath ++ baseName) imageType
              let pageContents =
                    case exitCode of
                      ExitSuccess -> renderExampleForm exSpec (image ! [src latticeUrl, border 1])
expValChart
                                    where latticeUrl = "/" ++ tmpImgPath ++ baseName ++ "." ++
imageType
                      _ -> p << "renderEx: error generating lattice image"
              return $ renderExamplePage pageContents
      else return $ renderExamplePage $ renderExampleForm exSpec (prToTable pr)
expValChart
```

```
renderExDefault = renderExamplePage $
                  renderExampleForm (ExContr ("zcb", [fromIntegral t1Horizon, 10], True))
                                 noHtml noHtml

renderExamplePage contents = renderPage pageTitle $
      p ! [align "right"] << anchor ! [href "/Contracts.html"] << "Source code"
  +++ contents

renderPage :: (HTML a, HTML b) => a -> b -> Html
 renderPage hdg contents = (header << (styleSheet +++ thetitle << hdg))
                       +++ (body << (h1 << hdg +++ contents))

styleSheet :: Html
 styleSheet = thelink ! [rel "stylesheet", thetype "text/css", href "/contracts.css" ] << noHtml
```

evalEx evaluates the contract specified by ExContr. Instead of pattern matching on the ExContr here, to avoid duplication it uses the examples list which is used in generating the web form.

```
evalEx :: ExContr -> PR Double
 evalEx (ExContr (name, args, f)) =
  case lookup name examples of
    Just (desc, defaultArgs, f) -> if length args >= length defaultArgs -- ignore extra args
                               then f args         -- TODO: could handle argument defaulting here? See
getArg.
                               else dummyContract
    _ -> dummyContract
  where
    dummyContract = evalX $ zcb time0 0 USD -- TODO: proper error reporting (to web page
if appropriate)
```

Limit server abuse - disallow large lattices in web interface. The program can easily handle thousands of time steps, but generating a graphic of the resulting lattice produces large files and consumes CPU resources. To experiment with larger trees, run Contracts.lhs on your own machine.

```
sanitize r = min (truncate r) 20
```

Map an example id to a description, default arguments, and an evaluation function.

```
examples =
  -- Contracts
  [("zcb",   ("Zero-coupon bond",    [t1Horizon, 10],
                                   (\(r:x:_) -> evalX $ zcb (mkDate $ sanitize r) x USD)))
  ,("c11",   ("European option",     [],  (\_       -> evalX c11)))
  -- Underlyings
  ,("probs", ("Probability lattice", [9], (\(r:_)   -> let n = sanitize r + 1 in PR $ take n
probabilityLattice)))
  ,("rates", ("Interest rate model", [9], (\(r:_)   -> let n = sanitize r + 1 in takePr n $ rateModel
xm USD)))]

renderExampleForm (ExContr (contractId, args, showImage)) chart1 chart2 =
  form ! [method "GET", action "/contractEx"]
    << table << ((tr << (td << "Contract" +++ td << "Horizon" +++ td << "Value" +++ td <<
"Output"))
           +++ (tr << ((td $ select ! [name "contract"]
                               << (map (\(id, (desc, defaultArgs, _)) ->
                                    attrIf (id == contractId) selected (option ! [value id]) << desc)
                                  examples))
                  +++ (td << textfield "arg1" ! [value $ getArg contractId args 0, size "10"])
                  +++ (td << textfield "arg2" ! [value $ getArg contractId args 1, size "10"])
                  +++ (td << (attrIf     showImage  checked (radio "image" "True")  +++
"Image"))))
           +++ (tr  << (td << submit "submit" "Draw" +++ spacer 2
                  +++ td << (attrIf (not showImage) checked (radio "image" "False") +++
"Table"))))
    +++ chart1 +++ hr +++ chart2
```

Retrieve the nth argument from the argument array; if not present, retrieve from default args for specified example.

```
getArg id l n = if n < length l then show $ l !! n
              else case lookup id examples of
                   Just (_, args, _) -> if n < length args
                                    then show $ args !! n else ""
```

attrIf adds the specified attribute to the Html element if the condition is true. Useful for checked and selected attributes.

```
attrIf False attr el = el
 attrIf True  attr el = el ! [attr]
```

Appendix F - HAppS server integration

The following module can be used to integrate with the HAppS application server. This provides a web interface to the system.

This code is not an executable part of Contracts.lhs. To use it, it should be extracted to its own file and built with HAppS. The import Contracts line imports Contracts.lhs (this file).

```
module Main where
```

```
import HAppS.Server.AlternativeHTTP
import HAppS.Server.HTTP.AltFileServe
import Control.Monad.State
import Numeric

import Contracts

instance FromData ExContr where
  fromData = do c    <- look "contract"
                arg1 <- look "arg1"
                arg2 <- look "arg2"
                img  <- look "image"
                return $ ExContr (c, map fst $ readFloat arg1
                                   ++ readFloat arg2, read img)

main :: IO ()
main = do simpleHTTP [dir "contractEx"
                       [withData $ \(ExContr t) ->
                         [anyRequest $ liftIO $ do out <- renderEx (ExContr t)
                                                   return $ toResponse out]
                       ,anyRequest $ ok $ toResponse renderExDefault]
                     ,fileServe ["Contracts.html"] "public" -- fileserving
                     ]
```

References

1. Composing contracts: an adventure in financial engineering
2. How to write a financial contract
3. Functional Reactive Animation

Credits

First, thanks to the authors of the original papers - Simon Peyton Jones, Jean-Marc Eber, and Julian Seward - for some fascinating and useful papers.
Thanks also to Thomas Hartman, Jeff Polakow, Adam Peacock, Chung-Chieh Shan and the organizers and members of the New York Functional Programmers Meetup Group for their encouragement and support.

The HTML version of this document was generated from Contracts.lhs using hscolour with the -lit and -css options to color only the code fragments. Worked like a charm. (The rest of the HTML was written by hand.)

nofficial example implementation of the system originally described in the paper Composing contracts: an adventure in financial engineering, by Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Familiarity with both versions of this paper is assumed.
This example implementation is a literate Haskell program, Contracts.lhs. Don't be put off by the embedded HTML — it is an executable Haskell program. It can be executed directly and experimented with in an interactive Haskell environment such as GHCI.
The program is completely self contained, depending only on a few GHC libraries. Compatibility with other Haskell implementations has not been tested.
A web interface to some of the examples is also available.
Please note that this program is an example, intended only for educational use in its current form. The core implementation is only 215 lines of code, excluding examples and user interface. As such, it has many limitations. See the Future work section for further information.

Contents

A note about the original papers

There are a few significant differences between the first and second papers. The change with the most impact on the implementation relates to the representation of contract horizons.
In A:5.3, entitled "Implementation in Haskell", a value process representation is described in which the lattice for a value process is stored as a list of random variables "in reverse time order", i.e. horizon first, along with the horizon's timestep number. This is possible because the first paper uses a simple, definite approach to representing horizons: they are specified by a single date.
However, this implementation is not compatible with some of the design decisions described in the second paper (B). Since the horizon of a contract can depend on observables other than the date, the second paper introduces a more sophisticated approach to representing horizons: horizons are specified by boolean value processes (type PR Bool) that define contract acquisition regions.
This allows the horizon of a contract to cross over more than one time step, depending on the value of the observable(s) that define the horizon. The use of boolean value processes "to describe the 'region' in which one can acquire a contract" is described as a "breakthrough" in section 3.6 of the second paper.
Since contracts can have indefinite horizons, and also because some value processes may have no horizon, it's not possible in general to represent a value process horizon-first, as suggested in the first paper.

Since the second paper does not provide an explicit description of the Haskell representation, our implementation uses a variation on the one described in the first paper, with some changes to support the second paper's design.

Module header

```
module Contracts
 -- (renderEx, renderExDefault, ExContr(ExContr)) -- limit exports for use from HAppS
 where

import List
 import Numeric
 import Control.Monad
 import System
 import Text.XHtml.Strict
 import Data.Unique
```

Notational conventions

Most references in this document are of the form P:S.s, where e.g. A:3.5 refers to paper A, section 3.5. The papers are listed in the References section. Papers A and B are the first and second versions of the Composing Contracts paper, and paper C is a paper about the functional reactive programming system Fran.
Notational conventions from B:Fig.1:

```
c, d, u : Contract
      o : Observable
   t, s : Date, time
      k : Currency
      x : Dimensionless real value
      p : Value process
      v : Random variable
```

Basic data types

```
data Currency = USD | GBP | EUR | ZAR | KYD | CHF  deriving (Eq, Show)
```

A Date is represented as a pair consisting of the start date/time of a contract, represented by a value of type CalendarTime, and an integer representing the number of time steps since the start of a contract. For example purposes, the time steps are of unspecified duration, and the CalendarTime type is stubbed out.

Representing the time step as a separate integer is useful when manipulating trees representing the evolution of a process over time, where the time step corresponds to an index into a list.

```
type Date = (CalendarTime, TimeStep)

type TimeStep = Int
 type CalendarTime = ()
```

Since the example doesn't use real dates, mkDate cheats and creates a Date from a TimeStep.

```
mkDate :: TimeStep -> Date
 mkDate s = ((),s)
```

Because real dates aren't used, all value processes are assumed to begin at the same time, the zeroth time step, time0.

```
time0 :: Date
 time0 = mkDate 0
```

This simplifies some aspects of the implementation, discussed further under Support actual dates and times.

Contract implementation

The representation of a contract is based on A:5.3.

```
data Contract =
    Zero
  | One  Currency
  | Give Contract
  | And  Contract Contract
  | Or   Contract Contract
  | Cond    (Obs Bool)   Contract Contract
  | Scale   (Obs Double) Contract
  | When    (Obs Bool)   Contract
  | Anytime (Obs Bool)   Contract
  | Until   (Obs Bool)   Contract
  deriving Show
```

Observable data type

"In general, a value of type Obs *d* represents a time-varying quantity of type *d*." (A:3.3)
An obvious implementation might involve a function of type (Date -> a). However, a "quantity" in this context is not a single value, but rather a random variable, i.e. a set of possible values.

This suggests a function of type (Date -> RV a). This would allow an arbitrary observable to be converted to a value process during evaluation of a contract, by applying successive dates to the observable's function. This would be a valid implementation, but in order to take maximum advantage of the lazy list representation used for value processes, we will actually use the following type:

```haskell
newtype Obs a = Obs (Date -> PR a)
```

An observable is thus represented as a function from a starting date to a value process. The "time-varying" nature of an observable is captured primarily by the value process itself (PR a); the Date in the function's type is simply used to specify the start date for the resulting value process.

For development and debugging purposes, Obs will be showable. Since it is a function type, this is achieved by applying it to a dummy date and displaying the first slice of the resulting process. This is only useful for getting a rough idea of the definition of a contract. See Enhance observable representation for further discussion of this.

```haskell
instance Show a => Show (Obs a) where
  show (Obs o) = let (PR (rv:_)) = o time0 in "(Obs " ++ show rv ++ ")"
```

Primitives for Defining Contracts

Define a combinator interface to the Contract datatype. From B:Fig.2:

```haskell
zero :: Contract
 zero = Zero

one :: Currency -> Contract
 one = One

give :: Contract -> Contract
 give = Give

cAnd :: Contract -> Contract -> Contract
 cAnd = And

cOr :: Contract -> Contract -> Contract
 cOr = Or

cond :: Obs Bool -> Contract -> Contract -> Contract
 cond = Cond

scale :: Obs Double -> Contract -> Contract
 scale = Scale
```

```
cWhen :: Obs Bool -> Contract -> Contract
 cWhen = When

anytime :: Obs Bool -> Contract -> Contract
 anytime = Anytime

cUntil :: Obs Bool -> Contract -> Contract
 cUntil = Until
```

Derived combinators

Other combinators can now be derived from these primitives, e.g.:

```
andGive :: Contract -> Contract -> Contract
 andGive c d = c `cAnd` give d
```

Primitives over observables

konst x is an observable that has value x at any time.

```
konst :: a -> Obs a
 konst k = Obs (\t -> bigK k)
```

lift f o is the observable whose value is the result of applying f to the value of the observable o.

```
lift :: (a -> b) -> Obs a -> Obs b
 lift f (Obs o) = Obs (\t -> PR $ map (map f) (unPr $ o t))
```

$lift_2$ o_1 o_2 is the observable whose value is the result of applying f to the values of the observables o_1 o_2.

```
lift2 :: (a -> b -> c) -> Obs a -> Obs b -> Obs c
 lift2 f (Obs o1) (Obs o2) = Obs (\t -> PR $ zipWith (zipWith f) (unPr $ o1 t) (unPr $ o2 t))
```

"The value of the observable date at date t is just t."

```
date :: Obs Date
 date = Obs (\t -> PR $ timeSlices [t])
```

"All numeric operations lift to the Obs type. The implementation is simple, using lift and $lift_2$."

```
instance Num a => Num (Obs a) where
   fromInteger i = konst (fromInteger i)
   (+) = lift2 (+)
   (-) = lift2 (-)
   (*) = lift2 (*)
```

```
  abs = lift abs
  signum = lift signum
```

One quirk is that we need to define a stub for Eq to support the Num instance.

```
instance Eq a => Eq (Obs a) where
  (==) = undefined
```

We can't implement Eq on an Observable's function, but we can provide a lifted version of equality:

```
(==*) :: Ord a => Obs a -> Obs a -> Obs Bool
 (==*) = lift2 (==)
```

at is a boolean observable that becomes True at time t (B:3.2)

```
at :: Date -> Obs Bool
 at t = date ==* (konst t)
```

Typeclasses don't work so well for relational operators, so define a separate family of them (B:3.3)

```
(%<), (%<=), (%=), (%>=), (%>) :: Ord a => Obs a -> Obs a -> Obs Bool
 (%<)  = lift2 (<)
 (%<=) = lift2 (<=)
 (%=)  = lift2 (==)
 (%>=) = lift2 (>=)
 (%>)  = lift2 (>)
```

Option contracts

From B:3.4:

```
european :: Date -> Contract -> Contract
 european t u = cWhen (at t) (u `cOr` zero)

american :: (Date, Date) -> Contract -> Contract
 american (t1, t2) u = anytime (between t1 t2) u

between :: Date -> Date -> Obs Bool
 between t1 t2 = lift2 (&&) (date %>= (konst t1)) (date %<= (konst t2))
```

Value processes

A value process PR a is represented as a list of random variables RV a, with the random variable corresponding to the earliest time step appearing first in the list.

```
newtype PR a = PR { unPr :: [RV a] } deriving Show
```

Note that the "informal type definition" of a value process is given in B:4.1 as `PR a = Date -> RV a`. However, this definition should not be taken literally. Among other things, it is not amenable to efficient list-based recursive processing of entire value processes, since it requires a lookup for access to each successive date. (This was discovered the hard way in an earlier implementation of this code — thanks to Chung-chieh Shan for pointing out the advantages of relying pervasively on a lazy list implementation, during the presentation of the earlier version of this code in NYC.)
A random variable `RV a` describes the possible values for a value process at a particular time step. For example, the random variable describing the outcome of a dice throw would be[1,2,3,4,5,6]. Random variables are therefore implemented as simple lists.

```
type RV a = [a]
```

Value process helpers

`takePr` truncates a (possibly infinite) value process.

```
takePr :: Int -> PR a -> PR a
 takePr n (PR rvs) = PR $ take n rvs
```

`horizonPr` determines the number of time steps in a value process. Only terminates for finite value processes.

```
horizonPr :: PR a -> Int
 horizonPr (PR rvs) = length rvs
```

`andPr` returns True if every value in a value process is true, false otherwise. Only terminates for finite value processes.

```
andPr :: PR Bool -> Bool
 andPr (PR rvs) = and (map and rvs)
```

Model

The model specifies the particular semantics for underlying observables such as the evolution of interest rates, exchange rates, and the types of calculation used. The contract evaluation function, evalC, is parameterized over a model to allow different models to be easily used. The model itself is implemented as a record of model-specific data and functions which can easily be instantiated by a function such as `exampleModel` below. Essentially, this amounts to a poor man's higher-order module.

```
data Model = Model {
   modelStart :: Date,
   disc       :: Currency -> (PR Bool, PR Double) -> PR Double,
   exch       :: Currency -> Currency -> PR Double,
   absorb     :: Currency -> (PR Bool, PR Double) -> PR Double,
   rateModel  :: Currency -> PR Double
   }
```

Define a specific model which defines the model functions given in the paper. This would normally be defined in a separate module.

```
exampleModel :: CalendarTime -> Model
 exampleModel modelDate = Model {
  modelStart = (modelDate,0),
  disc       = disc,
  exch       = exch,
  absorb     = absorb,
  rateModel  = rateModel
  }

 where
```

The example model's functions are defined in the following sections. Note that these definitions are local to the exampleModel record definition (due to the where clause above).

Interest rate model

See B:5.1. This constructs a lattice containing possible interest rates given a starting rate and an increment per time step. This "unrealistically regular" model matches that shown in B:Fig.8. However, it is so simple that some interest rates go negative after a small number of time steps. A better model is needed for real applications. Don't use this to model your retirement fund!

```
 rates :: Double -> Double -> PR Double
  rates rateNow delta = PR $ makeRateSlices rateNow 1
    where
      makeRateSlices rateNow n = (rateSlice rateNow n) : (makeRateSlices (rateNow-delta)
(n+1))
      rateSlice minRate n = take n [minRate, minRate+(delta*2) ..]
```

Each currency has different parameters for the interest rate model. Since the model is not realistic, these parameters are currently entirely arbitrary.

```
 rateModels = [(CHF, rates 7   0.8)
              ,(EUR, rates 6.5 0.25)
              ,(GBP, rates 8   0.5)
              ,(KYD, rates 11  1.2)
              ,(USD, rates 5   1)
              ,(ZAR, rates 15  1.5)
```

```
            ]

 rateModel k =
    case lookup k rateModels of
      Just x -> x
      Nothing -> error $ "rateModel: currency not found " ++ (show k)
```

'Disc' primitive

The primitive (disc t k) maps a real-valued random variable at date T, expressed in currency k, to its "fair" equivalent stochastic value process in the same currency k. See B:4.4 and B:Fig.7.
A simplifying assumption is that at some point, the boolean-valued process becomes True for an entire RV. This provides a simple termination condition for the discounting process.

```
 disc :: Currency -> (PR Bool, PR Double) -> PR Double
  disc k (PR bs, PR rs) = PR $ discCalc bs rs (unPr $ rateModel k)

   where

     discCalc :: [RV Bool] -> [RV Double] -> [RV Double] -> [RV Double]
      discCalc (bRv:bs) (pRv:ps) (rateRv:rs) =
        if and bRv -- test for horizon
          then [pRv]
          else let rest@(nextSlice:_) = discCalc bs ps rs
                   discSlice = zipWith (\x r -> x / (1 + r/100)) (prevSlice nextSlice) rateRv
                   thisSlice = zipWith3 (\b p q -> if b then p else q) -- allow for partially discounted slices
                               bRv pRv discSlice
               in thisSlice : rest
```

prevSlice calculates a previous slice in a lattice by averaging each adjacent pair of values in the specified slice

```
     prevSlice :: RV Double -> RV Double
      prevSlice [] = []
      prevSlice (_:[]) = []
      prevSlice (n1:rest@(n2:_)) = (n1+n2)/2 : prevSlice rest
```

'Absorb' primitive

"Given a boolean-valued process o, the primitive absorb$_k$(o,p) transforms the real-valued process p, expressed in currency k, into another real-valued process. For any state, the result is the expected value of receiving p's value if the region o will never be True, and receiving zero in the contrary. In states where o is True, the result is therefore zero."

```
 absorb :: Currency -> (PR Bool, PR Double) -> PR Double
  absorb k (PR bSlices, PR rvs) =
    PR $ zipWith (zipWith $ \o p -> if o then 0 else p)
                   bSlices rvs
```

Exchange rate model

This is a stub which always returns 1.

```
 exch :: Currency -> Currency -> PR Double
  exch k1 k2 = PR (konstSlices 1)
```

The definition of the exampleModel ends here.

Expected value

The code for absorb above does not obviously deal with the expected value mentioned in the spec. This is because the expected value of each random variable is implicit in the value process lattice representation: each node in the lattice is associated with a probability, and the expected value at a particular date is simply the sum of the product of the value at each node and its associated probability. The following functions implement this calculation.

```
expectedValue :: RV Double -> RV Double -> Double
 expectedValue outcomes probabilities = sum $ zipWith (*) outcomes probabilities

expectedValuePr :: PR Double -> [Double]
 expectedValuePr (PR rvs) = zipWith expectedValue rvs probabilityLattice
```

Snell primitive

Not currently implemented. The paper describes the following as a possible algorithm:

- take the final column of the tree (horizon),
- discount it back one time step,
- take the maximum of that column with the corresponding column of the original tree,
- then repeat that process all the way back to the root.

$\text{snell}_k(o,p)$ is the smallest process q (under an ordering relation mention briefly at the end of B:4.6) such that:

forall o' . (o => o') => q >= $\text{snell}_k(o',q)$

That is, an American option is the least upper bound of any of the deterministic acquisition choices specified by o', where o' is a sub-region of o.

Probability calculation

Each node in a value process lattice is associated with a probability.
"...in our very simple setting the number of paths from the root to the node is proportional to the probability that the variable will take that value."

```
probabilityLattice :: [RV Double]
 probabilityLattice = probabilities pathCounts
   where

    probabilities :: [RV Integer] -> [RV Double]
     probabilities (sl:sls) = map (\n -> (fromInteger n) / (fromInteger (sum sl))) sl : probabilities
sls
```

To calculate the number of paths to each node in a lattice, simply add the number of paths to the pair of parent nodes. This needs to work with Integers as opposed to Ints, because:
findIndex (\sl -> maximum sl > (fromIntegral (maxBound::Int))) pathCounts ==> Just 67

```
    pathCounts :: [RV Integer]
     pathCounts = paths [1] where paths sl = sl : (paths (zipWith (+) (sl++[0]) (0:sl)))
```

Compositional valuation semantics for contracts

See B:Fig.4. A Haskell type signature for eval is specified in A:5.3. It has been modified here to return a PR Double, as specified in the semantics in Figure 4, instead of a ValProc. (In this implementation, the PR Double type is essentially equivalent to the first paper's ValProc type.)

```
evalC :: Model -> Currency -> Contract -> PR Double
 evalC (Model modelDate disc exch absorb rateModel) k = eval    -- punning on record
fieldnames for conciseness
   where eval Zero           = bigK 0
         eval (One k2)       = exch k k2
         eval (Give c)       = -(eval c)
         eval (o `Scale` c)  = (evalO o) * (eval c)
         eval (c1 `And` c2)  = (eval c1) + (eval c2)
         eval (c1 `Or` c2)   = max (eval c1) (eval c2)
         eval (Cond o c1 c2) = condPr (evalO o) (eval c1) (eval c2)
         eval (When o c)     = disc   k (evalO o, eval c)
--       eval (Anytime o c)  = snell  k (evalO o, eval c)
         eval (Until o c)    = absorb k (evalO o, eval c)
```

Valuation semantics for observables

See B:Fig.5. The evalation function for observables, evalO, converts an observable's function to a value process by applying the function to a start date.

```
evalO :: Obs a -> PR a
 evalO (Obs o) = o time0
```

Process primitives

B:Fig6

```
bigK :: a -> PR a
 bigK x = PR (konstSlices x)

konstSlices x = nextSlice [x]
   where nextSlice sl = sl : (nextSlice (x:sl))

datePr :: PR Date
 datePr = PR $ timeSlices [time0]

timeSlices sl@((s,t):_) = sl : timeSlices [(s,t+1) | _ <- [0..t+1]]

condPr :: PR Bool -> PR a -> PR a -> PR a
 condPr = lift3Pr (\b tru fal -> if b then tru else fal)

liftPr :: (a -> b) -> PR a -> PR b
 liftPr f (PR a) = PR $ map (map f) a

lift2Pr :: (a -> b -> c) -> PR a -> PR b -> PR c
 lift2Pr f (PR a) (PR b) = PR $ zipWith (zipWith f) a b

lift2PrAll :: (a -> a -> a) -> PR a -> PR a -> PR a
 lift2PrAll f (PR a) (PR b) = PR $ zipWithAll (zipWith f) a b

lift3Pr :: (a -> b -> c -> d) -> PR a -> PR b -> PR c -> PR d
 lift3Pr f (PR a) (PR b) (PR c) = PR $ zipWith3 (zipWith3 f) a b c
```

A version of zipWith that handles input lists of different lengths. This is used to support lifted binary operations such as (+).

```
zipWithAll :: (a -> a -> a) -> [a] -> [a] -> [a]
 zipWithAll f (a:as) (b:bs)     = f a b : zipWithAll f as bs
 zipWithAll f as@(_:_) []       = as
 zipWithAll f []       bs@(_:_) = bs
 zipWithAll _ _        _        = []

instance Num a => Num (PR a) where
   fromInteger i = bigK (fromInteger i)
```

```
  (+) = lift2PrAll (+)
  (-) = lift2PrAll (-)
  (*) = lift2PrAll (*)
  abs = liftPr  abs
  signum = liftPr signum

instance Ord a => Ord (PR a) where
  max = lift2Pr max

instance Eq a => Eq (PR a) where
  (PR a) == (PR b) = a == b
```

Examples

Instantiate the example model with what would be the model's starting date, if real dates were used.

```
xm :: Model
 xm = exampleModel ()
```

Define an evaluator specific to the example model and the USD currency.

```
evalX :: Contract -> PR Double
 evalX = evalC xm USD
```

The by-now-infamous zero-coupon bond:

```
zcb :: Date -> Double -> Currency -> Contract
 zcb t x k = cWhen (at t) (scale (konst x) (one k))
```

A contract using the ZCB:

```
c1 :: Contract
 c1 = zcb t1 10 USD
```

The test date for the bond is `horizon` timesteps from the model's current date:

```
t1 :: Date
 t1 = mkDate t1Horizon

t1Horizon = 3 :: TimeStep
```

A stripped-down versions of the European option from B:3.4. That example uses real dates that range over more than two years. This is a smaller version. Its results have not been checked.

```
c11 :: Contract
 c11 = european (mkDate 2)
```

```
         (zcb (mkDate 20) 0.4 USD `cAnd`
          zcb (mkDate 30) 9.3 USD `cAnd`
          zcb (mkDate 40) 109.3 USD `cAnd`
          give (zcb (mkDate 12) 100 USD))
```

Evaluate the contract c1, in dollars, to produce a value process:

```
pr1 :: PR Double
 pr1 = evalX c1
```

Access the underlying lattice (list of slices):

```
tr1 = unPr pr1
```

Test of 'cUntil' - implementation of absorbEx is similar to zcb, but uses cUntil instead of cWhen.

```
absorbEx t x k = cUntil (konst t %> date) (scale (konst x) (one k))
```

Main function

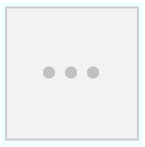
There is no main function. This program is intended to be run in an interactive Haskell environment such as GHCI, where the above examples, combinators, and evaluation functions can be examined and experimented with.
A web interface to some of the examples is also available.

Pretty pictures

Process lattice for zcb

The following diagram shows the value process lattice for the contract (zcb (mkDate 3) 10 USD). It matches B:Fig.9, except for minor details such as the fetching shade of pink. It was generated using GraphViz, by the following code.

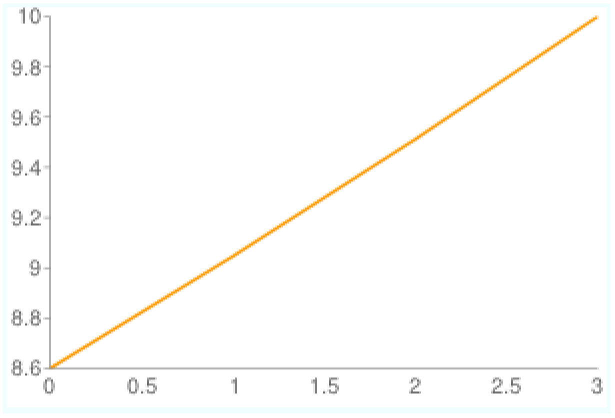
```
zcbImage = latticeImage pr1 "fig9" "png"
```



Expected value

The following is a chart of expected value at each timestep. The implementation uses the Google Chart API.

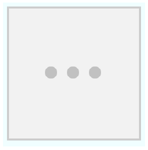
```
c1ExpectedValueUrl = chartUrl $ expectedValuePr pr1
```



Interest rate evolution

This is the short-term interest rate evolution from B:Fig8.

```
rateEvolution = latticeImage (takePr (t1Horizon + 1) $ rateModel xm USD) "fig8" "png"
```



Future work

In its present state, this implementation is very much a prototype, intended mainly to provide a concrete illustration of concepts described in the papers on which it is based. The core implementation is less than 200 lines of Haskell code, excluding examples and user interface. As such, it has many limitations and omissions. Some of the more obvious enhancements that could be made include:

- Support actual dates and times.
- Implement monadic contract evaluator.
- Enhance observable representation.
- Improve contract horizon handling.
- Implement more realistic interest and exchange rate models.
- Implement the snell function.
- Implement a model using other kinds of numerical methods, such as Monte Carlo.
- Implement models for specific observables.
- Factoring into modules.

Additional detail for selected items follows.

Support actual dates and times

As described in Basic data types, time is currently modeled using abstract integer time steps. Adding basic support for real dates and times should not be difficult. Since many financial contracts do not need to be concerned with time steps smaller than days, the following

description will focus only on date handling, but the same design applies to support for real times.
Adding date support requires changes in the following areas:

- Value processes should store their start date in order to correctly handle operations involving processes that start at different dates.
- Operations on value processes need to check the start dates of the processes they operate on, and proceed accordingly. For example, when adding two value processes that begin at different dates, only random variables which represent the same time step should be added together.
- Functions that rely on the zeroth time step time0, particularly the observable evalution function evalO and the date process datePr, should instead use the start date of the contract evaluator'sModel parameter. This can be achieved by converting the contract evaluator evalC from direct style to monadic style (see next subsection), which will allow the functions in question access to the model's start date.

Implement monadic contract evaluator

The evaluator for contracts is currently a very simple, direct-style implementation. This is possible in part because of simplifying choices such as the use of time steps without actual dates, as mentioned above.
Many other kinds of enhancements to the implementation are likely to require a more sophisticated evaluator design. Converting to a monadic evaluator would support such enhancements. Aside from providing primitives direct access to the model, it would also allow alternate monads and monad transformers to be used to parameterize the evaluator semantics. Jeff Polakow pointed out that a probability monad could be useful, for example.

Enhance observable representation

Two ways in which the representation of observables might be enhanced are as follows:

- C:4.1 suggests that the type could be designed to support simplification and other changes to the observable over time, as well as efficient handling of time intervals (C:4.2). The details here depend on the requirements of the system being developed.
- As the definition of the Show instance for the Observable data type demonstrates, the representation of observables as functions limits the ability to inspect contract definitions. With an embedded DSL, if metadata is not stored along with the DSL terms, then the host language source code may be the only complete specification of embedded terms that involve functions. This could be addressed by the use of a tag to identify observables. This might take the form of a sum type representing primitive observables such as constants and dates, with provision for arbitrary named functions for more complex observables. Such a representation is hinted at in the description of the valuation semantics for observables in B:Fig.5.

Improve contract horizon handling

The second paper's use of boolean value processes to represent acquisition regions is very general. The full generality of this model is not exploited by the current implementation. For example, thedisc primitive assumes that the horizon of a contract corresponds to a single random variable. However, contract horizons may be based on more complex observables than the date, and composed contracts may also result in complex scenarios in which a contract's horizon crosses more than one random variable.
In addition, some value processes are infinite and have no horizons. The system should track this to allow it to prevent attempts to perform non-terminating operations on such contracts. This may also help in implementing operations that combine contracts with different horizons.

Appendix A - Tests

A few small tests that came up during development.

```
tolerance = 0.001
```

Test of constant process:

```
testK = andPr $ liftPr (== 100) $ takePr 10 (bigK 100)
```

Test that a slice in the probability lattice adds up to probability 1.0:

```
testProb = (sum $ probabilityLattice !! 100) - 1 < tolerance
```

Test the result of evaluating the c1 contract

```
testPr1 = andPr $ lift2Pr (\a b -> (abs (a - b)) < tolerance)
                    pr1
                    (PR [[8.641], [9.246,8.901], [9.709,9.524,9.346], [10,10,10,10]])
```

Run all tests (all three of them!)

```
tests = and [testK
            ,testProb
            ,testPr1]
```

Appendix B - HTML table output for value process

This renders a value process lattice as a kind of pyramid, using an HTML table.

```
prToTable pr@(PR rvs) = table << (snd $ foldl renderSlice (0, noHtml) rvs)
  where

   horizon = horizonPr pr
    renderSlice (n, rows) rv = (n+1, rows +++ (tr $ td << (show n)
                                    +++ (spacer $ horizon - n)
                                    +++ (concatHtml (map renderCell rv))
                                    +++ (spacer $ horizon - n + 1)))

   renderCell v = td ! [theclass "cell", colspan 2] << (showFFloat (Just 2) v "")

spacer 0 = noHtml
 spacer n = td ! [theclass "sp", colspan n] << noHtml
```

Appendix C - Graphviz Output

This code generates graphs which represent a value process lattice. Currently assumes Double values, constrained by showNode's formatting of the value.
Write out tree as Dot file and run Dot to generate image:

```
latticeImage :: PR Double -> String -> String -> IO ExitCode
 latticeImage pr baseName imageType =
   do writeTreeAsDot baseName pr
      runDot baseName imageType
```

Supports interactive display of generated Dot code.

```
printTree :: PR Double -> IO ()
 printTree pr = mapM_ putStrLn (dotGraph (prToDot pr))
```

Write a value process out as a Dot file.

```
writeTreeAsDot :: String -> PR Double -> IO ()
 writeTreeAsDot baseName pr = writeFile (baseName ++ dotExt) $ unlines (dotGraph
(prToDot pr))
```

Run Dot on a file with the specified base name, and generate a graphic file with the specified type.

```
runDot :: String -> String -> IO ExitCode
 runDot baseName fileType =
   system $ concat ["dot -T", fileType,
                    " -o ", baseName, ".", fileType, " ",
                    baseName, dotExt]
```

Convert a (PR Double) to a list of dot node relationships.

```
prToDot :: PR Double -> [String]
 prToDot (PR rvs) = rvsToDot rvs
```

Convert lattice to list of dot node relationships.

```
rvsToDot :: [RV Double] -> [String]
 rvsToDot rvs = let numberedRvs = assignIds rvs 1
```

```
                in showNodes numberedRvs ++ treeToDot numberedRvs
```

```
dotExt = ".dot"
```

Number each of the nodes in a lattice.

```
assignIds :: [RV a] -> Int -> [RV (Int, a)]
 assignIds [] n = []
 assignIds (rv:rvs) n = numberList (reverse rv) n : assignIds rvs (n + length rv)
```

```
numberList :: [a] -> Int -> [(Int, a)]
 numberList l n = zip [n .. n + length l] l
```

showNodes returns a list of "primary" Dot representations of numbered RV nodes, with each node's value specified as the node's label. These nodes can then be referenced repeatedly in the generated Dot code without specifying a label.

```
showNodes :: [RV (Int, Double)] -> [String]
 showNodes numberedRvs = concatMap showSlice (numberList numberedRvs 0)
   where showSlice (n, sl) = ("subgraph Slice" ++ show n ++ " { rank=same")
                             : (map (\(n,s) -> show n ++ nodeLabel s) sl)
                             ++ ["SL" ++ (show n) ++ " [label=\"" ++ show n ++ "\" style=solid
peripheries=0] }"]
```

```
nodeLabel :: Double -> String
 nodeLabel s = " [label=\"" ++ (showFFloat (Just 2) s "\"]")
```

generate Dot code for relationships between numbered RV nodes.

```
treeToDot :: [RV (Int, a)] -> [String]
 treeToDot [a] = []
 treeToDot (a:b:rest) = dotJoin a (take (length a) b)
                     ++ dotJoin a (tail b)
                     ++ treeToDot (b:rest)
```

```
dotJoin :: RV (Int, a) -> RV (Int, a) -> [String]
 dotJoin a b = zipWith (\(m,a) (n,b) -> (show m) ++ " -- " ++ (show n)) a b
```

```
dotGraph :: [String] -> [String]
 dotGraph body = dotGraphHdr ++ (map formatDotStmt body) ++ ["}"]
```

```
dotGraphHdr :: [String]
 dotGraphHdr = ["graph contract_lattice {"
              ,"  rankdir=LR;"
              ,"  dir=none;"
              ,"  node [style=filled color=pink shape=box fontsize=10 width=0.5 height=0.4];"]

formatDotStmt :: String -> String
 formatDotStmt s = "  " ++ s ++ ";"
```

Appendix D - Google chart

This generates a URL for the Google Chart API. Used for expected value chart.

```
chartUrl :: [Double] -> String
 chartUrl vs =
"http://chart.apis.google.com/chart?chs=300x200&cht=lc&chxt=x,y&chg=20,25,2,5&chxr=0,0,"
            ++ (show $ length vs - 1)
            ++ "|1," ++ (showFFloat (Just 1) ymin ",")
                   ++ (showFFloat (Just 1) ymax "&chd=t:")
            ++ (concat $ intersperse "," $ map (\y -> showFFloat (Just 1) y "") ys)
  where (ymin, ymax, ys) = chartScale vs 100
```

Scale specified list of values to a range between 0 and upper.

```
chartScale ys upper =
  let ymin = minimum ys
      ymax = maximum ys
      yrange = ymax - ymin
      yscale = upper/yrange
  in (ymin, ymax, map (\y -> (y - ymin) * yscale ) ys)
```

Appendix E - Web interface

The following code implements a very simple web interface, which allows a few canned examples to be run and displays the resulting value process lattice images along with a chart of expected value (where appropriate).
At the time of writing, the web interface is running at http://contracts.scheming.org/contractEx. This URL may change in future.

Serializable example specification

ExContr is a type to specify examples to be run, which is serialized in the request URL.

```
newtype ExContr = ExContr (String, [Double], Bool) deriving (Read,Show,Eq)

useLatticeImage (ExContr (_, _, b)) = b

 webPath = "/home/anton/usr/happs.org/public/"
 -- webPath = "/home/anton/happs91/HAppS-Begin/public/"

 tmpImgPath = "imgtmp/"

 baseDotFilename = "pr-lattice"

 pageTitle = "Composing contracts - simple charts"

mkUniqueName :: String -> IO String
 mkUniqueName baseName =
   do u <- newUnique
      return $ baseName ++ (show $ hashUnique u)

renderEx :: ExContr -> IO Html
 renderEx exSpec@(ExContr (contractId, args, lattice)) =
   let pr = evalEx exSpec
       expValChart = if contractId == "probs" then noHtml -- expected value is meaningless for
the probabilities it relies on
                     else h3 << "Expected value" +++ image ! [src (chartUrl $ expectedValuePr pr)]
       imageType = "png"
   in if useLatticeImage exSpec
      then do baseName <- mkUniqueName baseDotFilename
              exitCode <- latticeImage pr (webPath ++ tmpImgPath ++ baseName) imageType
              let pageContents =
                    case exitCode of
                      ExitSuccess -> renderExampleForm exSpec (image ! [src latticeUrl, border 1])
expValChart
                                      where latticeUrl = "/" ++ tmpImgPath ++ baseName ++ "." ++
imageType
                      _ -> p << "renderEx: error generating lattice image"
              return $ renderExamplePage pageContents
      else return $ renderExamplePage $ renderExampleForm exSpec (prToTable pr)
expValChart

renderExDefault = renderExamplePage $
                     renderExampleForm (ExContr ("zcb", [fromIntegral t1Horizon, 10], True))
                                       noHtml noHtml

renderExamplePage contents = renderPage pageTitle $
       p ! [align "right"] << anchor ! [href "/Contracts.html"] << "Source code"
   +++ contents
```

```
renderPage :: (HTML a, HTML b) => a -> b -> Html
 renderPage hdg contents = (header << (styleSheet +++ thetitle << hdg))
                          +++ (body << (h1 << hdg +++ contents))
```

```
styleSheet :: Html
 styleSheet = thelink ! [rel "stylesheet", thetype "text/css", href "/contracts.css" ] << noHtml
```

evalEx evaluates the contract specified by ExContr. Instead of pattern matching on the ExContr here, to avoid duplication it uses the examples list which is used in generating the web form.

```
evalEx :: ExContr -> PR Double
 evalEx (ExContr (name, args, f)) =
   case lookup name examples of
     Just (desc, defaultArgs, f) -> if length args >= length defaultArgs -- ignore extra args
                                    then f args          -- TODO: could handle argument defaulting here? See
getArg.
                                    else dummyContract
     _ -> dummyContract
   where
     dummyContract = evalX $ zcb time0 0 USD -- TODO: proper error reporting (to web page
if appropriate)
```

Limit server abuse - disallow large lattices in web interface. The program can easily handle thousands of time steps, but generating a graphic of the resulting lattice produces large files and consumes CPU resources. To experiment with larger trees, run Contracts.lhs on your own machine.

```
sanitize r = min (truncate r) 20
```

Map an example id to a description, default arguments, and an evaluation function.

```
examples =
   -- Contracts
   [("zcb",   ("Zero-coupon bond",    [t1Horizon, 10],
                                       (\(r:x:_) -> evalX $ zcb (mkDate $ sanitize r) x USD)))
   ,("c11",   ("European option",     [],  (\_       -> evalX c11)))
   -- Underlyings
   ,("probs", ("Probability lattice", [9], (\(r:_)   -> let n = sanitize r + 1 in PR $ take n
probabilityLattice)))
   ,("rates", ("Interest rate model", [9], (\(r:_)   -> let n = sanitize r + 1 in takePr n $ rateModel
xm USD)))]
```

```
renderExampleForm (ExContr (contractId, args, showImage)) chart1 chart2 =
   form ! [method "GET", action "/contractEx"]
     << table << ((tr << (td << "Contract" +++ td << "Horizon" +++ td << "Value" +++ td <<
"Output"))
           +++ (tr << ((td $ select ! [name "contract"]
                             << (map (\(id, (desc, defaultArgs, _)) ->
                                      attrIf (id == contractId) selected (option ! [value id]) << desc)
                                   examples))
                 +++ (td << textfield "arg1" ! [value $ getArg contractId args 0, size "10"])
                 +++ (td << textfield "arg2" ! [value $ getArg contractId args 1, size "10"])
                 +++ (td << (attrIf      showImage  checked (radio "image" "True")  +++
"Image"))))
           +++ (tr  << (td << submit "submit" "Draw" +++ spacer 2
                  +++ td << (attrIf (not showImage) checked (radio "image" "False") +++
"Table"))))
    +++ chart1 +++ hr +++ chart2
```

Retrieve the nth argument from the argument array; if not present, retrieve from default args for specified example.

```
getArg id l n = if n < length l then show $ l !! n
             else case lookup id examples of
                  Just (_, args, _) -> if n < length args
                                  then show $ args !! n else ""
```

attrIf adds the specified attribute to the Html element if the condition is true. Useful for checked and selected attributes.

```
attrIf False attr el = el
 attrIf True  attr el = el ! [attr]
```

Appendix F - HAppS server integration

The following module can be used to integrate with the HAppS application server. This provides a web interface to the system.
This code is not an executable part of Contracts.lhs. To use it, it should be extracted to its own file and built with HAppS. The import Contracts line imports Contracts.lhs (this file).

```
module Main where

import HAppS.Server.AlternativeHTTP
import HAppS.Server.HTTP.AltFileServe
import Control.Monad.State
import Numeric

import Contracts

instance FromData ExContr where
```

```
 fromData = do c    <- look "contract"
              arg1 <- look "arg1"
              arg2 <- look "arg2"
              img  <- look "image"
              return $ ExContr (c, map fst $ readFloat arg1
                                      ++ readFloat arg2, read img)

main :: IO ()
main = do simpleHTTP [dir "contractEx"
                     [withData $ \(ExContr t) ->
                       [anyRequest $ liftIO $ do out <- renderEx (ExContr t)
                                                 return $ toResponse out]
                     ,anyRequest $ ok $ toResponse renderExDefault]
                   ,fileServe ["Contracts.html"] "public" -- fileserving
                   ]
```

References

1. Composing contracts: an adventure in financial engineering
2. How to write a financial contract
3. Functional Reactive Animation

Credits

First, thanks to the authors of the original papers - Simon Peyton Jones, Jean-Marc Eber, and Julian Seward - for some fascinating and useful papers.
Thanks also to Thomas Hartman, Jeff Polakow, Adam Peacock, Chung-Chieh Shan and the organizers and members of the New York Functional Programmers Meetup Group for their encouragement and support.
The HTML version of this document was generated from Contracts.lhs using hscolour with the -lit and -css options to color only the code fragments. Worked like a charm. (The rest of the HTML was written by hand.)