Aivika 3:
Creating a Simulation Library based on Functional Programming

David E. Sorokin <david.sorokin@gmail.com>,
Yoshkar-Ola, Mari El, Russia

January 24, 2015

Abstract

This paper describes a unified formalism that can be applied for modeling and simulating a wide range of dynamic systems which are usually studied in the Discrete Event Simulation and System Dynamics disciplines. The suggested formalism uses the functional programming approach and its practical usability essentially depends on whether and how the programming language supports a syntactic sugar for monadic and arrow computations. The formalism is implemented in the open-source Aivika simulation library available on the official Hackage repository complementing the Haskell Platform.

Contents

1 Introduction **2**

2 Simulation **3**

3 External Parameters **4**

4 Ordinary Differential Equations **5**

5 Event-oriented Simulation **9**

6 Mutable Reference **10**

7 Example: Event-oriented Simulation **10**

8 Variable with Memory **11**

9 Process-oriented Simulation **12**

10 Example: Process-oriented Simulation **15**

11 Activity-oriented Simulation **17**

12 Example: Activity-oriented Simulation **17**

13 Signaling **19**

14 Queue Strategies **21**

15 Resource **22**

16 Example: Using Resources **23**

17 Queues **25**

18 Stream **26**

19 Processor **28**

20 Server **31**

21 Example: Queue Network **31**

22 Agent-based Modeling **34**

23 Example: Agent-based Modeling **35**

24 Circuit **37**

25 Network **39**

26 Simulation Experiment **40**

27 Example: Parametric Model **41**

28 Example: Using Arrays **44**

29 Related Work **45**

30 Conclusions **46**

1 Introduction

Many real processes occurring in economics, politics and industry can be modeled by *dynamic systems*: those that vary in time.

Such systems can be described using various paradigms:

- System Dynamics (SD);
- Discrete Event Simulation (DES);
- Agent-based Modeling (AM).

A computer *simulation* allows running numeric experiments to attempt to reproduce the behavior of such systems. It can help us to optimize the real underlying processes and try to understand and realize their internal mechanisms. Also the experiments allow providing the 'what-if' case scenarios. The role of the computer simulation is especially important if the real experiments are too costly, impractical or even impossible.

An idea of combining different approaches to modeling and simulation was always attracting.

For example, there is a commercial simulation software tool AnyLogic[25] that allows the modeler to combine System Dynamics, DES and Agent-based Modeling within one model.

In the course of studying functional programming the author, being a professional software developer with a mathematical background, was interested in that how the functional programming could be applied to simulation.

As a result, the author created the Aivika[23] simulation library, which formalism is represented here. It uses the Haskell programming language.

There are other implementations of the Aivika library made by the author. Historically, the first one[18] that was published is written in the F# programming language. There is also a Scala version[17]. But without any doubt, the Haskell version of Aivika is the most superior one.

The library is quite different in comparison with other Haskell simulation libraries such as Yampa[12, 11, 13] and Hydra[4, 3].

2 Simulation

We can treat the simulation as a polymorphic function of the simulation run:

```
newtype Simulation a = Simulation (Run -> IO a)
```

It is obvious that the `Simulation` type is an instance[1] of the `MonadFix` and `MonadIO` type classes. The former allows defining the recursive monadic computations, while the latter allows embedding any `IO` computation in the simulation.

Given the specified simulation `Specs`, we can create a simulation `Run`, which type is defined below, and then launch the simulation itself to receive the result:

```
runSimulation :: Simulation a -> Specs -> IO a
```

The simulation specs can contain the information about the start time and final time of modeling. Since the represented formalism also allows us to integrate the differential equations, we must provide the integration time step and the method regardless of whether they are actually used. Also the specs can define the random number generator that we can use in the model.

```
data Specs = Specs { spcStartTime :: Double,
                     -- ^ the start time
                     spcStopTime :: Double,
                     -- ^ the stop time
                     spcDT :: Double,
                     -- ^ the integration time step
                     spcMethod :: Method,
                     -- ^ the integration method
```

[1]As some readers noted, the `Simulation` monad could indeed be defined with help of the `ReaderT` monad transformer, but that would be too inefficient. Since the simulation must be as fast as possible and Aivika is mainly designed to be a practical tool, the simulation library uses its own implementation of this monad. The same is related to other computations defined in Aivika.

```
                      spcGeneratorType :: GeneratorType
                      -- ^ the type of the random number generator
                    }
```

The mentioned `Run` type is quite implementation dependent and it is hidden in depths of API from the direct using by the modeler. At least, it must contain the specs provided so that they could be passed in to every part of the `Simulation` computation.

```
data Run = Run { runSpecs :: Specs,
                 -- ^ the simulation specs
                 runIndex :: Int,
                 -- ^ the current simulation run index
                 runCount :: Int,
                 -- ^ the total number of runs in this experiment
                 runEventQueue :: EventQueue,
                 -- ^ the event queue
                 runGenerator :: Generator
                 -- ^ the random number generator
               }
```

To allow running the Monte-Carlo simulation, the `Run` value must also contain the information about how many simulation runs are launched in parallel as well as it must contain the current simulation run index.

Then we can run the specified number of parallel simulations, where each simulation run will be distinguished by its index as well as it will contain its own instances of the event queue and random number generator:

```
runSimulations :: Simulation a -> Specs -> Int -> [IO a]
```

The main idea is that many simulation models can be ultimately reduced to the `Simulation` computation. Hence they can be trivially simulated using the mentioned above run functions by the specified specs.

3 External Parameters

In practice many models depend on external parameters, which is useful for providing the Sensitivity Analysis.

To represent such parameters, we can use almost the same definition that we used for representing the `Simulation` computation.

```
newtype Parameter a = Parameter (Run -> IO a)
```

A key difference between these two computations is that the parameter can be memoized before running the simulation so that the resulting `Parameter` computation would return a constant value within every simulation run and then its value would be updated for other runs (in a thread-safe way).

```
memoParameter :: Parameter a -> IO (Parameter a)
```

We usually have to memoize the parameter if its computation is not pure and it depends on the `IO` actions such as reading an external file or generating a random number.

It is natural to represent the simulation specs as external parameters when modeling.

```
starttime :: Parameter Double
stoptime :: Parameter Double
dt :: Parameter Double
```

Since we provide the random number generator with the simulation specs, it is also natural to generate the random numbers within the `Parameter` computation.

```
randomUniform :: Double -> Double -> Parameter Double
randomNormal :: Double -> Double -> Parameter Double
randomExponential :: Double -> Parameter Double
randomErlang :: Double -> Int -> Parameter Double
randomPoisson :: Double -> Parameter Int
randomBinomial :: Double -> Int -> Parameter Int
```

To support the Design of Experiments (DoE), we specify two additional computations that just return the corresponding fields from the simulation `Run` defining the current simulation run index and the total run count respectively.

```
simulationIndex :: Parameter Int
simulationCount :: Parameter Int
```

An arbitrary parameter can be lifted into the `Simulation` computation with help of the following function, which is actually defined in Aivika with help of a type class.

```
class ParameterLift m where
  liftParameter :: Parameter a -> m a

instance ParameterLift Simulation
```

It allows using the parameter within the simulation.

4 Ordinary Differential Equations

Assuming that the `Point` type represents a modeling time point within the current simulation run, we can define a polymorphic time varying function which would be suitable for approximating the integrals.

Let us call it the `Dynamics` computation to emphasize that it can model some dynamic processes defined usually with help of ordinary differential equations (ODEs) and difference equations of System Dynamics.

```
newtype Dynamics a = Dynamics (Point -> IO a)
```

The `Dynamics` type is obviously an instance of the `MonadFix` and `MonadIO` type classes too.

Since the modeling time is passed in to every part of the `Dynamics` computation, it is natural to define the following computation that would return the current time.

```
time :: Dynamics Double
```

There are different functions that allow running the `Dynamics` computation within the simulation: in the start time, in the final time, in all integration time points and in arbitrary time points defined by their numeric values.

```
runDynamicsInStartTime :: Dynamics a -> Simulation a
runDynamicsInStopTime :: Dynamics a -> Simulation a
runDynamicsInIntegTimes :: Dynamics a -> Simulation [IO a]

runDynamicsInTime :: Double -> Dynamics a -> Simulation a
runDynamicsInTimes :: [Double] -> Dynamics a -> Simulation [IO a]
```

A key feature of the `Dynamics` computation is that it allows approximating the integral by the specified derivative and initial value:

```
integ :: Dynamics Double -> Dynamics Double -> Simulation (Dynamics Double)
```

The second parameter of the function might be a pure value, but using a computation here is more useful for practice as it allows referring to the initial value of the integral when defining the differential equations.

The point is that the ordinary differential and difference equations can be defined declaratively almost as in maths and as in many commercial simulation software tools of System Dynamics such as Vensim[27], ithink/Stella[8], Berkeley-Madonna[9] and Simtegra MapSys[2].

To create an integral, we have to allocate an internal array to store the approximated values in the integration time points. We perform this side effect within the `Simulation` computation, where we have an access to the simulation specs.

Moreover, we can make the parameterized `Dynamics a` type numeric if type `a` is numeric, following the approach described in [5] by Paul Hudak. It means that we can treat the `Dynamics Double` value as a number.

```
instance (Num a) => Num (Dynamics a)
```

To demonstrate the approach, we can rewrite a model from the 5-Minute Tutorial of Berkeley-Madonna[9] with the following equations.

$$\begin{aligned}
\dot{a} &= -ka \times a, \qquad a(t_0) = 100, \\
\dot{b} &= ka \times a - kb \times b, \qquad b(t_0) = 0, \\
\dot{c} &= kb \times b, \qquad c(t_0) = 0, \\
ka &= 1, \\
kb &= 1.
\end{aligned}$$

For example, we can return the integral values in the final simulation time. In the same way, we could return the integral values in arbitrary time points that we would specify using other run functions.

```
model :: Simulation [Double]
model =
  mdo a <- integ (- ka * a) 100
      b <- integ (ka * a - kb * b) 0
      c <- integ (kb * b) 0
      let ka = 1
          kb = 1
      runDynamicsInStopTime $ sequence [a, b, c]
```

[2]In the past the author developed visual simulation software tool `Simtegra MapSys`, but the software is unfortunately not available for the wide audience at time of preparing this article anymore.

Here we base on the fact that the `Simulation` type is `MonadFix` and hence it supports the recursive *do*-notation.

The difference equations can be defined in a similar manner. The reader can find an example in the Aivika distribution.

Regarding the arrays and vectors of integrals, they are created naturally in Haskell. No special support is required. Only we need to use also the recursive *do*-notation to define an array if it has a loopback. The corresponded example is provided in the Aivika distribution too.

It is worth noting that we can embed external functions in the differential equations. It is possible thanks to the fact that the `Simulation` and `Dynamics` types are monads. The equations are defined as the corresponding computations which we can bind with the external functions using the *do*-notation.

Moreover, there are helper functions that allow embedding many external functions having a side effect. These helper functions order the calculations in the integration time points and use an interpolation in other time points.

For example, one of these functions is used in the mentioned above `integ` function for integrating.

```
memoDynamics :: Dynamics e -> Simulation (Dynamics e)
memo0Dynamics :: Dynamics e -> Simulation (Dynamics e)
```

The both functions memoize and order the resulting `Dynamics` computation in the integration time points. When requesting for a value in another time point, the both functions apply the simplest interpolation returning the value calculated in the closest less integration time point. But the functions behave differently when integrating the equations with help of the Runge-Kutta method.

The `Point` type contains the additional information to distinguish intermediate integration time points used by the method. While the `memoDynamics` function memoizes the values in these intermediate time points, the second function `memo0Dynamics` just ignore these points applying the interpolation.

Therefore, the first memoization function is used by the `integ` function. In all other cases the second memoization function is more preferable as it is more efficient and consumes less memory.

Regarding the `Point` type, it is implementation-dependent. Like the `Run` type it is hidden from the direct using by the modeler. The definition may change in the future without affecting the rest API.

```
data Point = Point { pointSpecs :: Specs,
                     -- ^ the simulation specs
                     pointRun :: Run,
                     -- ^ the simulation run
                     pointTime :: Double,
                     -- ^ the current time
                     pointIteration :: Int,
                     -- ^ the current iteration
                     pointPhase :: Int
                     -- ^ the current phase
                   }
```

Here the `pointPhase` field allows Aivika to distinguish the integration time points from the intermediate integration time points used by the Runge-Kutta method and from all other time points used in the discrete event simulation.

The considered above simulation monads are imperative as they are based on the `IO` monad. Namely this fact allows using the random number generator within the simulation. Therefore, the ordinary differential equations can be stochastic. The Aivika library provides useful helper random functions similar to those ones that are used in other software tools of System Dynamics.

```
memoRandomUniformDynamics ::
  Dynamics Double -> Dynamics Double -> Simulation (Dynamics Double)

memoRandomNormalDynamics ::
  Dynamics Double -> Dynamics Double -> Simulation (Dynamics Double)

memoRandomExponentialDynamics ::
  Dynamics Double -> Simulation (Dynamics Double)

memoRandomErlangDynamics ::
  Dynamics Double -> Dynamics Int -> Simulation (Dynamics Double)

memoRandomPoissonDynamics ::
  Dynamics Double -> Simulation (Dynamics Int)

memoRandomBinomialDynamics ::
  Dynamics Double -> Dynamics Int -> Simulation (Dynamics Int)
```

They are based on the mentioned earlier random functions returning the `Parameter` computation, but only these functions memoize the generated values in the integration time points and apply the interpolation in all other time points. The functions are designed to be used in the differential and difference equations.

Regarding the difference equations themselves, they can be built like differential ones. The following function returns an accumulated sum represented as the `Dynamics` computation by the specified difference and initial value.

```
diffsum :: (Num a, Unboxed a)
           => Dynamics a
           -> Dynamics a
           -> Simulation (Dynamics a)
```

Here the `Unboxed` type class specifies what types can represent *unboxed* values for more efficient storing in memory. Its definition can be found in the Aivika library.

Since the `Point` type refers to the current simulation `Run`, we can lift an arbitrary `Simulation` computation into the `Dynamics` one.

```
class SimulationLift m where
  liftSimulation :: Simulation a -> m a

instance SimulationLift Dynamics
instance ParameterLift Dynamics
```

It literally means that we can use the external parameters and computations defined on level of the simulation run in the ordinary differential and difference equations.

Now let us turn from the differential and difference equations of System Dynamics to the realm of simulation models ruled by discrete events.

5 Event-oriented Simulation

Under the *event-oriented* paradigm[16, 10] of DES, we put all pending events in the priority queue, where the first event has the minimal activation time. Then we sequentially activate the events removing them from the queue. During such an activation we can add new events. This scheme is also called *event-driven*.

We can use almost the same time-varying function for the event-oriented simulation, which we used for approximating the integrals with help of the `Dynamics` monad.

```
newtype Event a = Event (Point -> IO a)
```

The difference is that we can strongly guarantee[3] on level of the type system of Haskell that the `Event` computation is always synchronized with the event queue. Here we imply that every simulation run has an internal event queue, which the reader could notice in the definition of the `Run` type.

A key feature of the `Event` monad is an ability to specify the event handler that should be actuated at the desired modeling time, when the corresponding event occurs.

```
enqueueEvent :: Double -> Event () -> Event ()
```

To pass in a message or some other data to the event handler, we just use a closure when specifying the event handler in the second argument.

The event cancellation can be implemented trivially. We create a wrapper for the source event handler and pass in namely this wrapper to the `enqueueEvent` function. Then the wrapper already decides whether it should call the underlying source event handler. Then we have to provide some means for notifying the wrapper that the source event handler must be cancelled. The Aivika library has the corresponded support.

The same technique of canceling the event can be adapted to implementing the timer and time-out handlers used in the agent-based modeling as it is described later.

To involve in the simulation, the `Event` computation must be run explicitly or implicitly within the `Dynamics` computation. The most simple run function is stated below. It actuates all pending event handlers from the event queue relative to the current modeling time and then runs the specified computation.

```
runEvent :: Event a -> Dynamics a
```

There is a subtle thing related to the `Dynamics` computation. In general, the modeling time flows unpredictably within `Dynamics`, while there is a guarantee that the time is synchronized with the event queue within the `Event` computation.

Some other run functions are destined for the most important use cases, when we can run the input computation directly within `Simulation` in the initial and final modeling time points, respectively.

```
runEventInStartTime :: Event a -> Simulation a
runEventInStopTime :: Event a -> Simulation a
```

[3]Actually, there is a room in Aivika for some hacking that may break this strong guarantee.

Following the rule, an arbitrary `Dynamics` computation can be lifted into the `Event` computation. As before, the corresponding function is defined in a type class.

```
class DynamicsLift m where
  liftDynamics :: Dynamics a -> m a

instance DynamicsLift Event
instance SimulationLift Event
instance ParameterLift Event
```

It means that the integrals, external parameters and computations on level of the simulation run can be directly used in the event-oriented simulation.

6 Mutable Reference

Many DES models need a mutable reference. Since Haskell is a pure functional programming language, all side effects must be expressed explicitly in the type signatures. Mutable references require such effects.

In the Haskell standard libraries, `IORef` is the standard mutable reference. Aivika introduces a very similar but strict `Ref` reference, where all computations are synchronized with the event queue.

```
data Ref a

newRef :: a -> Simulation (Ref a)

readRef :: Ref a -> Event a
writeRef :: Ref a -> a -> Event ()
modifyRef :: Ref a -> (a -> a) -> Event ()
```

The simulation model should use the `Ref` type instead of `IORef` within the `Simulation` computation whenever possible.

7 Example: Event-oriented Simulation

The Aivika distribution contains examples of using the mutable references in the DES models, one of which is provided below. The task itself is described in the documentation of SimPy[10].

> There are two machines, which sometimes break down. Up time is exponentially distributed with mean 1.0, and repair time is exponentially distributed with mean 0.5. There are two repairpersons, so the two machines can be repaired simultaneously if they are down at the same time. Output is long-run proportion of up time. Should get value of about 0.66.

The corresponding model is as follows.

```
import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
```

```
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 1000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation Double
model =
  do totalUpTime <- newRef 0.0

     let machineBroken :: Double -> Event ()
         machineBroken startUpTime =

           do finishUpTime <- liftDynamics time
              modifyRef totalUpTime (+ (finishUpTime - startUpTime))
              repairTime <-
                liftParameter $
                randomExponential meanRepairTime

              -- enqueue a new event
              let t = finishUpTime + repairTime
              enqueueEvent t machineRepaired

         machineRepaired :: Event ()
         machineRepaired =

           do startUpTime <- liftDynamics time
              upTime <-
                liftParameter $
                randomExponential meanUpTime

              -- enqueue a new event
              let t = startUpTime + upTime
              enqueueEvent t $ machineBroken startUpTime

     runEventInStartTime $
       do -- start the first machine
          machineRepaired
          -- start the second machine
          machineRepaired

     runEventInStopTime $
       do x <- readRef totalUpTime
          y <- liftParameter stoptime
          return $ x / (2 * y)

main = runSimulation model specs >>= print
```

Frankly speaking, the use of the event-oriented paradigm may seem to be quite tedious. Aivika supports more high-level paradigms. Later it will be shown how the same task can be solved in a more simple way.

8 Variable with Memory

Sometimes we need an analog of the mutable reference that would save the history of its values. Aivika defines the corresponded `Var` type. It has almost the same functions with similar type signatures that the `Ref` reference has.

```
data Var a

newVar :: a -> Simulation (Var a)

readVar :: Var a -> Event a
writeVar :: Var a -> a -> Event ()
modifyVar :: Var a -> (a -> a) -> Event ()
```

However, we can also use the variable in the differential and difference equations requesting for the *first* actual value for *each* time point with help of the following function, actuating the pending events if required.

```
varMemo :: Var a -> Dynamics a
```

The magic is as follows. The `Var` variable stores the history of changes. When updating the mutable variable, or requesting it for a value at new time point, the `Var` data object stores internally the value, which was first for the requested time point. Then it becomes constant within the simulation. Therefore, the computation returned by the `varMemo` function can be used in the differential and difference equations of System Dynamics.

On the contrary, the `readVar` function returns a computation of the *recent* actual value for the *current* simulation time point. This value is already destined to be used in the discrete event simulation as it is synchronized with the event queue by the very design of the library. Such is the `Event` computation that it must be synchronized with the event queue.

In case of need we can freeze temporarily the variable and receive its internal state: triples of time, the first and last values for each time.

```
freezeVar :: Var a -> Event (Array Int Double, Array Int a, Array Int a)
```

The time values returned by this function are distinct and sorted in ascending order.

9 Process-oriented Simulation

Under the *process-oriented* paradigm[16, 10], we model simulation activities with help of a special kind of processes. We can explicitly suspend and resume such processes. Also we can request for and release of the resources implicitly suspending and resuming the processes in case of need.

Aivika actually supports the process-oriented simulation on two different levels. A higher level which uses the infinite streams of data and processors that operate on these streams is considered further. Below is described a lower level, which is a foundation for the higher level, nevertheless.

To model a process, Aivika uses the following monad as a basis.

```
newtype Cont a = Cont ((a -> Event ()) -> Event ())
```

The reader can notice that this is a partial case of the standard monad transformer `ContT` parameterized by the `Event` computation.

It is well known that we can suspend the `ContT` computation and then resume later. This is one of the main features that distinguishes that monad. So is the `Cont` monad stated above, being a partial case.

Only it is worth noting that the actual definition of the `Cont` monad in Aivika is more complicated to allow catching the `IO` exceptions and canceling the process. Here the author implemented a monad, which is very similar to the F# async workflow from the corresponded .NET programming language.

Returning to the simulation, a key idea is that the value of type `Cont ()` can be reduced to a function of type `Event () -> Event ()`, which is actually an end part of the type signature for the `enqueueEvent` function mentioned above. That function enqueues a new event with the desired time of actuating the event handler.

It literally means that we can take an arbitrary computation of type `Cont ()`, suspend it and then resume it at another modeling time with help of the event queue.

This technique allows us to hold the process for the specified time interval. But sometimes we need to passivate the process for indefinite time so that another simulation activity could reactivate it later.

Therefore, we need some data structure to store the continuation that we would receive within the `Cont` computation. The process identifier `ProcessId` can play a role of this data structure.

```
data ProcessId
newProcessId :: Simulation ProcessId
```

Then a *discontinuous process* can be represented with help of the following computation.

```
newtype Process a = Process (ProcessId -> Cont a)
```

This is obviously a `Monad` and `MonadIO`. Unfortunately, this is not `MonadFix` as it is based on continuations.

We can run the process within the simulation with help of one of the next functions.

```
runProcess :: Process () -> Event ()
runProcessUsingId :: ProcessId -> Process () -> Event ()

runProcessInStartTime :: Process () -> Simulation ()
runProcessInStartTimeUsingId :: ProcessId -> Process () -> Simulation ()

runProcessInStopTime :: Process () -> Simulation ()
runProcessInStopTimeUsingId :: ProcessId -> Process () -> Simulation ()
```

If the process identifier is not specified then a new generated identifier is assigned when running the process. Every process has always its own unique identifier.

```
processId :: Process ProcessId
```

In case of need we can run a sub-process using another identifier.

```
processUsingId :: ProcessId -> Process a -> Process a
```

A characteristic feature of the `Process` monad is that the process can be hold for the specified time interval through the event queue, following the approach described above in this section.

```
holdProcess :: Double -> Process ()
```

Nevertheless, the hold process can be immediately interrupted and we can request for whether it indeed was interrupted. The information about this is stored until the next call of the `holdProcess` function.

```
interruptProcess :: ProcessId -> Event ()
processInterrupted :: ProcessId -> Event Bool
```

It is worth noting to say more about the types of computations returned by these functions. The `Event` type of the result means that the computation executes immediately and it cannot be interrupted. On the contrary, the `Process` type of the result means that the corresponding computation may suspend, even forever. This is very important for understanding.

To passivate the process for indefinite time to reactive it later, we can use the following functions.

```
passivateProcess :: Process ()
processPassive :: ProcessId -> Event Bool
reactivateProcess :: ProcessId -> Event ()
```

Every process can be immediately cancelled, which is important for modeling some activities.

```
cancelProcessWithId :: ProcessId -> Event ()
cancelProcess :: Process a
processCancelled :: ProcessId -> Event Bool
```

Sometimes we need to run an arbitrary sub-process with the specified timeout.

```
timeoutProcess :: Double -> Process a -> Process (Maybe a)
```

If the sub-process executes too long and exceeds the time limit, then it is immediately canceled and `Nothing` is returned within the `Process` computation. Otherwise; the computed result is returned right after it is received by the sub-process.

Every simulation computation we considered before can be lifted into the `Process` computation.

```
class EventLift m where
  liftEvent :: Event a -> m a

instance EventLift Process
instance DynamicsLift Process
instance SimulationLift Process
instance ParameterLift Process
```

It allows using the integrals and external parameters as well as updating the mutable references and variables within the process-oriented simulation. It allows combining the event-oriented and process-oriented simulation.

Another process can be forked and spawn on-the-fly. If that process is not related to the current parent process in any way, then we can run the second process within the `Event` computation and then lift the result into the `Process` computation. There is no need to add a special function. It is enough to have `liftEvent` and one of the `Process` run functions.

```
liftEvent $ runProcess (p :: Process ())
```

But if the life cycle of the child process must be bound up with the life cycle of the parent process so that they would be canceled in some order if required, then we should use one of the next functions.

```
spawnProcess :: Process () -> Process ()
spawnProcess = spawnProcessWith CancelTogether

spawnProcessWith :: ContCancellation -> Process () -> Process ()
```

Here the first argument of the second function specifies how two processes are bound.

```
data ContCancellation =
    | CancelTogether
    | CancelChildAfterParent
    | CancelParentAfterChild
    | CancelInIsolation
```

The stated above `timeoutProcess` function uses `spawnProcessWith` to run the specified sub-process within time-out.

Also an arbitrary number of the `Process` computations can be launched in parallel and we can await the completion of all the started sub-processes to return the final result.

```
processParallel :: [Process a] -> Process [a]
```

The `Process` computation can be memoized so that the resulting process would always return the same value within the simulation run regardless of that how often the process was requested repeatedly.

```
memoProcess :: Process a -> Simulation (Process a)
```

The continuations are known to be difficult for handling exceptions properly. Nevertheless, the author adapted successfully the F# `Async` approach and added the corresponded functions to handle arbitrary exceptions within `IO`.

```
catchProcess :: Exception e => Process a -> (e -> Process a) -> Process a
finallyProcess :: Process a -> Process b -> Process a
throwProcess :: IOException -> Process a
```

There are similar functions for handling the exceptions within all other simulation monads considerer in the document before.

Thus, the functions described in this section allow efficiently modeling quite complex activities. Nevertheless, the `Process` computation is still low-level. Aivika supports more high-level computations described further.

10 Example: Process-oriented Simulation

Let us return to the task that was solved in section 7 using the event-oriented paradigm. The problem statement is repeated here. It corresponds to the documentation of SimPy.

There are two machines, which sometimes break down. Up time is exponentially distributed with mean 1.0, and repair time is exponentially distributed with mean 0.5. There are two repairpersons, so the two machines can be repaired simultaneously if they are down at the same time. Output is long-run proportion of up time. Should get value of about 0.66.

Using the processes, we can solve the task in a more elegant way.

```
import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 1000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation Double
model =
  do totalUpTime <- newRef 0.0

     let machine :: Process ()
         machine =
           do upTime <-
                liftParameter $
                randomExponential meanUpTime
              holdProcess upTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)
              repairTime <-
                liftParameter $
                randomExponential meanRepairTime
              holdProcess repairTime
              machine

     runProcessInStartTime machine
     runProcessInStartTime machine

     runEventInStopTime $
       do x <- readRef totalUpTime
          y <- liftParameter stoptime
          return $ x / (2 * y)

main = runSimulation model specs >>= print
```

The reader can compare this model with the previous one. Conceptually, they do the same thing, use the same event queue and have the same behavior.

There is also another popular paradigm applied to the discrete event simulation. It usually gives more rough simulation results as we have to scale the modeling time. The next two sections show how Aivika supports that paradigm and how we can apply it to solve the same task.

11 Activity-oriented Simulation

Under the *activity-oriented* paradigm[16, 10] of DES, we break time into tiny increments. At each time point, we look around at all the activities and check for the possible occurrence of events. Sometimes this scheme is called *time-driven.*

An idea is that we can naturally represent the activity as an `Event` computation, which we will call periodically through the event queue.

```
enqueueEventWithTimes :: [Double] -> Event () -> Event ()
enqueueEventWithTimes ts e = loop ts
  where loop []       = return ()
        loop (t : ts) = enqueueEvent t $ e >> loop ts
```

We can also use another predefined function that does almost the same, but only it calls the specified computation directly in the integration time points specified by the simulation specs.

```
enqueueEventWithIntegTimes :: Event () -> Event ()
```

Being defined in such a way, the activity-oriented simulation can be combined with the event-oriented and process-oriented ones. There is the corresponded example in the Aivika distribution, where the pit furnace[16, 26] is modeled.

12 Example: Activity-oriented Simulation

To illustrate the activity-oriented paradigm, let us take our old task that was solved in section 7 using the event-oriented paradigm and in section 10 using the process-oriented paradigm of DES. The problem statement is repeated here again. It corresponds to the documentation of SimPy.

> There are two machines, which sometimes break down. Up time is exponentially distributed with mean 1.0, and repair time is exponentially distributed with mean 0.5. There are two repairpersons, so the two machines can be repaired simultaneously if they are down at the same time. Output is long-run proportion of up time. Should get value of about 0.66.

Now the model looks quite cumbersome. Moreover, we have to scale the modeling time. The time points at which the events occur are not precise any more.

```
import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 1000.0,
                spcDT = 0.05,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation Double
model =
  do totalUpTime <- newRef 0.0

     let machine :: Simulation (Event ())
         machine =
           do startUpTime <- newRef 0.0

              -- a number of iterations when
              -- the machine works
              upNum <- newRef (-1)

              -- a number of iterations when
              -- the machine is broken
              repairNum <- newRef (-1)

              -- create a simulation model
              return $
                do upNum’ <- readRef upNum
                   repairNum’ <- readRef repairNum

                   let untilBroken =
                         modifyRef upNum $ \a -> a - 1

                       untilRepaired =
                         modifyRef repairNum $ \a -> a - 1

                       broken =
                         do writeRef upNum (-1)
                            -- the machine is broken
                            startUpTime’ <- readRef startUpTime
                            finishUpTime’ <- liftDynamics time
                            dt’ <- liftParameter dt
                            modifyRef totalUpTime $
                              \a -> a +
                              (finishUpTime’ - startUpTime’)
                            repairTime’ <-
                              liftParameter $
                              randomExponential meanRepairTime
                            writeRef repairNum $
                              round (repairTime’ / dt’)

                       repaired =
                         do writeRef repairNum (-1)
                            -- the machine is repaired
                            t’  <- liftDynamics time
                            dt’ <- liftParameter dt
                            writeRef startUpTime t’
                            upTime’ <-
                              liftParameter $
                              randomExponential meanUpTime
                            writeRef upNum $
                              round (upTime’ / dt’)

                       result | upNum’ > 0      = untilBroken
                              | upNum’ == 0     = broken
                              | repairNum’ > 0  = untilRepaired
                              | repairNum’ == 0 = repaired
                              | otherwise       = repaired
                   result

     -- create two machines with type Event ()
     m1 <- machine
     m2 <- machine

     -- start the time-driven simulation of the machines
     runEventInStartTime $
       -- in the integration time points
       enqueueEventWithIntegTimes $
       do m1
          m2

     -- return the result in the stop time
     runEventInStopTime $
       do x <- readRef totalUpTime
          y <- liftParameter stoptime
          return $ x / (2 * y)

main = runSimulation model specs >>= print
```

Nevertheless, the activity-oriented paradigm can be exceptionally useful for modeling some parts that are difficult to represent based on other simulation paradigms.

13 Signaling

Functional Reactive Programming (FRP) is a popular approach for simulation. For example, it is used in the Yampa library mentioned in the introduction.

In the F# programming language FRP is usually associated [15] with the `Async` monad and `IObservable` interface. There is the same association in Aivika, where the author adapted the F# approach.

The following monoid represents a signal that notifies about occurring some condition.

```
data Signal a =
  Signal { handleSignal :: (a -> Event ()) -> Event DisposableEvent }
```

The `handleSignal` function takes a signal and its handler, subscribes the handler for receiving the signal values and then returns a nested computation of the `DisposableEvent` type that being applied unsubscribes the specified handler from receiving the signal.

The act of unsubscribing from the signal occurs in a time. Therefore, the nested computation returned has actually type `Event ()` hidden under the facade of the convenient type name. The act of subscribing also occurs in a time.

```
disposeEvent :: DisposableEvent -> Event ()
```

If we are not going to unsubscribe at all, then we can ignore the nested computation.

```
handleSignal_ :: Signal a -> (a -> Event ()) -> Event ()
```

We can treat the signals in a functional way, transforming or merging or filtering them with help of combinators like these ones.

```
emptySignal :: Signal a
mapSignal :: (a -> b) -> Signal a -> Signal b
filterSignal :: (a -> Bool) -> Signal a -> Signal a
merge2Signals :: Signal a -> Signal a -> Signal a
```

The Ref reference and Var variable provide signals that notify about changing their state.

```
 refChanged :: Ref a -> Signal a
 varChanged :: Var a -> Signal a
```

We can create an origin of the signal manually. Distinguishing the origin from the signal allows us to publish the signal with help of a pure function. But we must trigger the signal within a computation synchronized with the event queue, though.

```
data SignalSource a

newSignalSource :: Simulation (SignalSource a)

publishSignal :: SignalSource a -> Signal a
triggerSignal :: SignalSource a -> a -> Event ()
```

The reader may notice that the Cont monad and Signal monoid have a similar definition. Even probably, the Signal computation could be defined as a monad (using the event queue if required) but it seems to be not very practicable from the author's point of view.

A link with FRP is expressed by the following function that suspends the simulation process until a signal comes.

```
processAwait :: Signal a -> Process a
```

In Aivika there is an opposite transformation from the Process computation to a Signal value, but it is a little bit complicated as the process can be actually canceled or an IO exception can be raised within the simulation. The corresponded transformation is defined with help of the Task type.

```
runTask :: Process a -> Event (Task a)
```

Here we run the specified process in background and immediately return the corresponded task within the Event computation. Later we can request for the result of the underlying Process computation, whether it was finished successfully, or an IO exception had occurred or the computation was cancelled. Please refer to the Aivika documentation for detail.

Using signals, the mentioned earlier function timeoutProcess is implemented. It allows us to run a sub-process within the specified time-out. The function creates an internal signal source. The launched sub-process is trying to compute the result and in case of success it notifies the parent process, triggering the corresponded signal.

The signals are also used in the memoProcess function. It takes the specified process and returns a new resulting process that will always return the same value within the current simulation run regardless of that how many times that process will be started. The value is calculated only once for each simulation run but all other process instances await the signal about receiving the result.

The memoProcess function is important for implementation of the streams of data and processors that operate on them.

14 Queue Strategies

Before we proceed to more high level modeling constructs, we need to define the *queue strategies*[26] that prescribe how the competitive requests must be prioritized.

Starting with version 2.0, in Aivika the queue strategies are expressed in terms of type families, where each queue strategy instance may specify its own queue storage.

```
class QueueStrategy s where

  data StrategyQueue s :: * -> *

  newStrategyQueue :: s -> Simulation (StrategyQueue s i)
  strategyQueueNull :: StrategyQueue s i -> Event Bool
```

The first function creates a queue by the specified strategy. The second one tests whether the queue is empty.

The `DequeueStrategy` type class defines a strategy applied in the dequeueing operation, when there are competitive requests which cannot be fulfilled immediately. Therefore, we have to decide what request has a higher priority when dequeueing.

```
class QueueStrategy s => DequeueStrategy s where
  strategyDequeue :: StrategyQueue s i -> Event i
```

The `EnqueueStrategy` type class defines a strategy applied in the plain enqueueing operation, when there are competitive requests which cannot be fulfilled immediately too. We prioritize the request somehow adding it to the queue of requests.

```
class DequeueStrategy s => EnqueueStrategy s where
  strategyEnqueue :: StrategyQueue s i -> i -> Event ()
```

There is also a version of the enqueueing strategy that uses priorities.

```
class DequeueStrategy s => PriorityQueueStrategy s p | s -> p where
  strategyEnqueueWithPriority :: StrategyQueue s i -> p -> i -> Event ()
```

There are four predefined queue strategies in Aivika at present:

- FCFS (First Come - First Served), or FIFO (First In - First Out);
- LCFS (Last Come - First Served), or LIFO (Last In - First Out);
- SIRO (Service in Random Order);
- StaticPriorities (Using Static Priorities).

These strategies are implemented as data types having a single data constructor with the corresponded name.

```
data FCFS = FCFS deriving (Eq, Ord, Show)
data LCFS = LCFS deriving (Eq, Ord, Show)
data SIRO = SIRO deriving (Eq, Ord, Show)
data StaticPriorities = StaticPriorities deriving (Eq, Ord, Show)
```

Each type is an instance of the corresponding queue strategy.

```
instance EnqueueStrategy FCFS
instance EnqueueStrategy LCFS
instance EnqueueStrategy SIRO
instance PriorityQueueStrategy StaticPriorities Double
```

15 Resource

A *resource*[10] simulates something to be queued for, for example, the machine.

```
data Resource s
```

Here the parametric type `s` represents a queue strategy.

The simplest constructor allows us to create a new resource by the specified queue strategy and initial amount.

```
newResource :: QueueStrategy s => s -> Int -> Simulation (Resource s)
```

To acquire the resource, we can use the predefined functions like these ones:

```
requestResource :: EnqueueStrategy s => Resource s -> Process ()

requestResourceWithPriority ::
  PriorityQueueStrategy s p => Resource s -> p -> Process ()
```

Each of the both suspends the process in case of the resource deficiency until some other simulation activity releases the resource.

```
releaseResourceWithinEvent :: DequeueStrategy s => Resource s -> Event ()
```

There is also a more convenient version of the last function that works within the `Process` computation, but the provided function emphasizes the fact that releasing the resource cannot block the simulation process and this action is performed immediately.

```
releaseResource :: DequeueStrategy s => Resource s -> Process ()
```

We can request for the current available amount of the specified resource as well as request for its maximum possible amount and the strategy applied.

```
resourceCount :: Resource s -> Event Int
resourceMaxCount :: Resource s -> Maybe Int
resourceStrategy :: Resource s -> s
```

The second function returns an optional value indicating that the maximum amount could be unspecified when creating the resource.

```
newResourceWithMaxCount ::
  QueueStrategy s => s -> Int -> Maybe Int -> Simulation (Resource s)
```

By default, the maximum possible amount is set equaled to the initial amount specified when calling the first constructor `newResource`.

There are type synonyms for resources using the predefined queue strategies.

```
type FCFSResource = Resource FCFS
type LCFSResource = Resource LCFS
type SIROResource = Resource SIRO
type PriorityResource = Resource StaticPriorities
```

There are constructors that use these type synonyms. Some of these constructors are provided below.

```
newFCFSResource :: Int -> Simulation FCFSResource
newLCFSResource :: Int -> Simulation LCFSResource
newSIROResource :: Int -> Simulation SIROResource
newPriorityResource :: Int -> Simulation PiriorityResource
```

Finally, there is a helper `usingResource` function that acquires the resource, runs the specified `Process` computation and finally releases the resource regardless of whether the specified process was cancelled or an exception was raised.

```
usingResource :: EnqueueStrategy s => Resource s -> Process a -> Process a
usingResource r m =
  do requestResource r
     finallyProcess m $ releaseResource r
```

We can do the same with the resource using priorities.

```
usingResourceWithPriority ::
  PriorityQueueStrategy s p => Resource s -> p -> Process a -> Process a
usingResourceWithPriority r priority m =
  do requestResourceWithPriority r priority
     finallyProcess m $ releaseResource r
```

The both functions use the `finallyProcess` function that allows executing a finalization part within the computation whenever an exception might arise or the computation was canceled at all. The functions guarantee that the resource will be released in any case.

16 Example: Using Resources

To illustrates how the resources can be used for modeling, let us again take a task from the documentation of SimPy[24].

> Two machines, but sometimes break down. Up time is exponentially distributed with mean 1.0, and repair time is exponentially distributed with mean 0.5. In this example, there is only one repairperson, so the two machines cannot be repaired simultaneously if they are down at the same time.
>
> In addition to finding the long-run proportion of up time, let us also find the long-run proportion of the time that a given machine does not have immediate access to the repairperson when the machine breaks down. Output values should be about 0.6 and 0.67.

In Aivika we can solve this task in the following way.

```
import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 1000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation (Double, Double)
model =
  do -- number of times the machines have broken down
     nRep <- newRef 0

     -- number of breakdowns in which the machine
     -- started repair service right away
     nImmedRep <- newRef 0

     -- total up time for all machines
     totalUpTime <- newRef 0.0

     repairPerson <- newFCFSResource 1

     let machine :: Process ()
         machine =
           do upTime <-
                liftParameter $
                randomExponential meanUpTime
              holdProcess upTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)

              -- check the resource availability
              liftEvent $
                do modifyRef nRep (+ 1)
                   n <- resourceCount repairPerson
                   when (n == 1) $
                     modifyRef nImmedRep (+ 1)

              requestResource repairPerson
              repairTime <-
                liftParameter $
                randomExponential meanRepairTime
              holdProcess repairTime
              releaseResource repairPerson

              machine

     runProcessInStartTime machine
     runProcessInStartTime machine

     runEventInStopTime $
       do x <- readRef totalUpTime
          y <- liftParameter stoptime
          n <- readRef nRep
          nImmed <- readRef nImmedRep
          return (x / (2 * y),
                  fromIntegral nImmed / fromIntegral n)

main = runSimulation model specs >>= print
```

As before, this is a complete Haskell program, which can be launched with help of the `runghc` utility or compiled to a native code.

17 Queues

Sometimes we need a location in the network where entities wait for service[16]. They are modeled in Aivika by the finite and infinite queues.

For brevity, only the finite queue is considered here as this is a more difficult case.

```
data Queue si sm so a
```

It represents a queue using the specified strategies for enqueueing (input), `si`, internal storing (in memory), `sm`, and dequeueing (output), `so`, where the parametric type `a` denotes the type of items stored in the queue.

There are type synonyms for the most important use cases:

```
type FCFSQueue a = Queue FCFS FCFS FCFS a
type LCFSQueue a = Queue FCFS LCFS FCFS a
type SIROQueue a = Queue FCFS SIRO FCFS a
type PriorityQueue a = Queue FCFS StaticPriorities FCFS a
```

These synonyms use the FIFO strategy both for enqueueing and dequeueing operations, which seems to be reasonable for many cases. However, we can define a finite queue that would process all pending dequeueing requests, for example, in random (SIRO) or opposite (LIFO) order in case of enqueueing an item to the empty queue.

There are different functions that allow creating a new empty queue by the specified capacity.

```
newQueue
  :: (QueueStrategy si, QueueStrategy sm, QueueStrategy so)
  => si -> sm -> so -> Int -> Event (Queue si sm so a)

newFCFSQueue :: Int -> Event (FCFSQueue a)
newLCFSQueue :: Int -> Event (LCFSQueue a)
newSIROQueue :: Int -> Event (SIROQueue a)
newPriorityQueue :: Int -> Event (PriorityQueue a)
```

Unlike many other data structures, a queue is created within the `Event` computation as we have to know the current simulation time to start gathering the timing statistics about the queue size. The statistics is initiated at time of invoking the computation.

There are different enqueueing functions. The most simple one is provided below.

```
enqueue :: (EnqueueStrategy si, EnqueueStrategy sm, DequeueStrategy so)
           => Queue si sm so a -> a -> Process ()
```

It suspends the process if the finite queue is full. Therefore, this action is returned as the `Process` computation.

Also we can try to enqueue and if the queue is full then the item is lost.

```
enqueueOrLost :: (EnqueueStrategy sm, DequeueStrategy so)
                 => Queue si sm so a -> a -> Event Bool
```

This action cannot already suspend the simulation activity and hence it returns the `Event` computation of a flag indicating whether the item was successfully stored in the queue.

The simplest dequeueing operation suspends the process while the queue is empty. The result is the `Process` computation again.

```
dequeue :: (DequeueStrategy si, DequeueStrategy sm, EnqueueStrategy so)
           => Queue si sm so a -> Process a
```

Here the very type signatures specify whether the corresponded function may suspend the simulation activity, or the action is performed immediately.

There are similar enqueueing and dequeueing functions that allow specifying the priorities if the corresponded queue strategy supports them.

The queue has a lot of counters that are updated during the simulation. Actually, these counters are what we are mostly interested in.

For example, we can request the queue for its size and wait time.

```
queueCountStats :: Queue si sm so a -> Event (TimingStats Int)
queueWaitTime :: Queue si sm so a -> Event (SamplingStats Double)
```

Here `SamplingStats` is a relatively light-weight and immutable data type that comprises the statistics summary collected for the queue's wait time. It returns the average value, variance, deviation, minimum, maximum and the number of samples. The `TimingStats` provides additional information about the times at which the minimum and maximum values were gained. Also the timing statistics takes into account the time at which data are gathered.

Below is shown how the queues can be processed using more high level constructs (processors) that operate on the stream of data.

18 Stream

Many things become significantly more simple for reasoning and understanding after we introduce a concept of *infinite stream* of data distributed sequentially in the modeling time. This idea is very consonant with FRP.

```
newtype Stream a = Cons { runStream :: Process (a, Stream a) }
```

This is a kind of the *cons-cell*, where the cell is returned within the `Process` computation. It means that the stream data can be distributed in the modeling time and there can be time gaps between sequential data.

The streams themselves are well-known in the functional programming for a long time[1]. It is obvious that we can map, filter, transform the streams.

Now it is more interesting what new properties we can gain introducing the `Process` computation. At least, the `Stream` type is a monoid.

Passivating the underlying process forever[4], we receive a stream that never returns data.

```
emptyStream :: Stream a
```

Moreover, we can merge two streams applying the `FCFS` strategy when enqueueing the input data.

```
mergeStreams :: Stream a -> Stream a -> Stream a
```

Actually, the latter is a partial case of more general functions that allow concatenating the streams like a *multiplexor*.

[4]The underlying process can still be canceled, though.

```
concatStreams :: [Stream a] -> Stream a
concatStreams = concatQueuedStreams FCFS

concatQueuedStreams ::
  EnqueueStrategy s => s -> [Stream a] -> Stream a

concatPriorityStreams ::
  PriorityQueueStrategy s p => s -> [Stream (p, a)] -> Stream a
```

The functions use the resources to concatenate different infinite streams of data.

There is an opposite ability to split the input stream into the specified number of output streams like a *demultiplexor*. We have to do it to model a parallel work of services.

```
splitStream :: Int -> Stream a -> Simulation [Stream a]
splitStream = splitStreamQueueing FCFS

splitStreamQueueing ::
  EnqueueStrategy s
  => s -> Int -> Stream a -> Simulation [Stream a]

splitStreamPrioritising ::
  PriorityQueueStrategy s p
  => s -> [Stream p] -> Stream a -> Simulation [Stream a]
```

An implementation of the second function is provided below for demonstrating the approach. Only we need an auxiliary function that creates a new stream as a result of the repetitive execution of some process.

```
repeatProcess :: Process a -> Stream a
```

Here is the `splitStreamQueueing` function itself:

```
splitStreamQueueing s n x =
  do ref <- liftIO $ newIORef x
     res <- newResource s 1
     let reader =
           usingResource res $
           do p <- liftIO $ readIORef ref
              (a, xs) <- runStream p
              liftIO $ writeIORef ref xs
              return a
     return $ map (\i -> repeatProcess reader) [1..n]
```

A key idea is that many simulation models can be defined as a network of the `Stream` computations.

Such a network must have external input streams, usually random streams like these ones.

```
randomUniformStream :: Double -> Double -> Stream (Arrival Double)
randomNormalStream :: Double -> Double -> Stream (Arrival Double)
randomExponentialStream :: Double -> Stream (Arrival Double)
randomErlangStream :: Double -> Int -> Stream (Arrival Double)
randomPoissonStream :: Double -> Stream (Arrival Int)
randomBinomialStream :: Double -> Int -> Stream (Arrival Int)
```

Here a value of type `Arrival a` contains the modeling time at which the external event has arrived, the event itself of type `a` and the delay time which has passed from the time of arriving the previous event.

To process the input stream in parallel, we split the input with help of the `splitStream` function, process new streams in parallel and then concatenate the intermediate results into one output stream using the `concatStreams` function. Later will be provided the `processorParallel` function that does namely this.

To process the specified stream sequentially by some servers, we need a helper function that would read one more data item in advance, playing a role of the intermediate buffer between the servers.

```
prefetchStream :: Stream a -> Stream a
```

Now we need the moving force that would run the whole network of streams.

```
sinkStream :: Stream a -> Process ()
```

It infinitely reads data from the specified stream.

The next section shows how we can create an `Arrow` based on the `Stream` computation. To implement the arrow, we need some helper functions among which the most important one is next.

```
memoStream :: Stream a -> Simulation (Stream a)
memoStream (Cons s) =
  do p <- memoProcess $
          do ~(x, xs) <- s
             xs' <- liftSimulation $ memoStream xs
             return (x, xs')
     return (Cons p)
```

It memoizes the stream so that the resulting stream would always return the same data within the simulation run. The implementation essentially depends on the `memoProcess` function mentioned above.

19 Processor

Having a stream of data, it would be natural to operate on its transformation which we will call a *processor*:

```
newtype Processor a b = Processor { runProcessor :: Stream a -> Stream b }
```

Here the choice of this name was not arbitrary [6]. This type seems to be an `Arrow`, but the arrow can be interpreted as some kind of processor.

Only the author of the current document and Aivika did not prove whether the introduced type `Processor` is an arrow and satisfies to its laws, but it can indeed be an arrow. If it is true then we can use the *proc*-notation of Haskell to operate on the streams. But if it is still not true then this is not so important as we can operate on the streams directly and this approach seems quite useful in practice for defining the simulation models.

Meantime, an implementation of the `Arrow`-related type classes made by the author in Aivika is essentially based on work [14]. Only there is a difference. To implement, it was necessary to introduce the `memoStream` function mentioned above. We rely on this function to unzip streams and further use the same approach as it is described in the referenced work.

We can construct the processors directly from the streams. Omitting the obvious cases, we consider only the most important ones.

A new processor can be created by the specified handling function producing the `Process` computation.

```
arrProcessor :: (a -> Process b) -> Processor a b
```

Also we can use an accumulator to save an intermediate state of the processor. When processing the input stream and generating an output one, we can update the state.

```
accumProcessor :: (acc -> a -> Process (acc, b)) -> acc -> Processor a b
```

An arbitrary number of processors can be united to work in parallel using the default `FCFS` queue strategy:

```
processorParallel :: [Processor a b] -> Processor a b
```

Its implementation is based on using the multiplexing an demultiplexing functions considered before. We split the input stream, process the intermediated streams in parallel and then concatenate the resulting streams into one output steam.

There are other versions of the `processParallel` function, where we can specify the queue strategies and priorities if required.

To create a sequence of autonomously working processors, we can use the prefetching function considered above too:

```
prefetchProcessor :: Processor a a
prefetchProcessor = Processor prefetchStream
```

For example, having two complementing processors `p1` and `p2`, we can create two new processors, where the first one implies a parallel work, while another implies a sequential processing:

```
pPar = processorParallel [p1, p2]
pSeq = p1 >>> prefetchProcessor >>> p2
```

Basing on this approach, we can model in Haskell quite complex *queue networks* in an easy-to-use high-level declarative manner, which makes the Aivika library close enough to some specialized simulation software tools.

Regarding the queues themselves, we can model them using rather general-purpose helper processors like this one:

```
queueProcessor :: (a -> Process ())
                  -- ^ enqueue the input item
                  -> Process b
                  -- ^ dequeue an output item
                  -> Processor a b
                  -- ^ the buffering processor
```

An idea is that there is a plenty of cases how the queues could be united in the network. When enqueueing, we can either wait while the queue is full, or we can count such an item as lost. We can use the priorities for the `Process` computations that enqueue or dequeue. Moreover, different processes can enqueue and dequeue simultaneously.

Therefore, the author decided to introduce such general-purpose helper functions for modeling the queues, where the details of how the queues are simulated can be shortly described with help of combinators like `enqueue`, `enqueueOrLost` and `dequeue` stated above. The examples are included in the Aivika distribution.

Unfortunately, the `Processor` type is not `ArrowLoop` by the same reason why the `Process` monad is not `MonadFix` — continuations. Nevertheless, we can model the queue networks with loopbacks using the intermediate queues to delay the stream. One of the possible functions is provided below.

```
queueProcessorLoopSeq ::
  (a -> Process ())
  -- ^ enqueue the input item
  -> Process c
  -- ^ dequeue an item for the further processing
  -> Processor c (Either e b)
  -- ^ process and then decide what values of type @e@
  -- should be processed in the loop (condition)
  -> Processor e a
  -- ^ process in the loop and then return a value
  -- of type @a@ to the queue again (loop body)
  -> Processor a b
  -- ^ the buffering processor
```

An example model that would use the streams and queue processors is provided further in section 21.

Using the processors, we can model a complicated enough behavior, for example, we can model the Round-Robbin strategy[26] of the processing.

```
roundRobbinProcessor :: Processor (Process Double, Process a) a
```

It tries to perform a task within the specified timeout. If the task times out, then it is canceled and returned to the processor again; otherwise, the successful result is redirected to output. The timeout and task are passed in to the processor from the input stream.

Both the processors and streams allow modeling the process-oriented simulation on a higher level in a way somewhere similar to that one which is described in book [16] by A. Alan B. Pritsker and Jean J. O'Reilly.

At the same time, all computations are well integrated in Aivika and we can combine different approaches within the same model, for example, lifting an arbitrary `Dynamics` computation such as an integral into the high-level `Processor` computation.

By the way, each time we use the modeling time and other simulation parameters such as the start time of final time, we use the same lifting functions that do exactly the same thing that they do, when lifting the integral. There is no difference.

Such is an essence of the approach suggested in this work, where the simulation computations are just functions, but the monads and arrow are an easy-to-use practical mechanism to build complex models from simple parts. Therefore, the syntax sugar provided by the Haskell compiler for creating monadic and arrow computations plays a very significant role and essentially determines how useful can be the approach in real practice.

20 Server

In Aivika there is a helper `Server` data type that allows modeling a working place and gathering its statistics.

```
data Server s a b

newServer :: (a -> Process b) -> Simulation (Server () a b)
newStateServer :: (s -> a -> Process (s, b)) -> s -> Simulation (Server s a b)
```

To create a server, we provide a handling function that takes the input, process it and generates an output within the `Process` computation. The handling function may use an accumulator to save the server state when processing.

To involve the server in simulation, we can use its processor that performs a service and updates the internal counters.

```
serverProcessor :: Server s a b -> Processor a b
```

For example, when preparing the simulation results to output, we can request for the statistics of the time spent by the server while processing the tasks.

```
serverProcessingTime :: Server s a b -> Event (SamplingStats Double)
```

There is one subtle thing. Each time we use the `serverProcessor` function, we actually create a new processor that refers to the same server and hence updates the same statistics counters. It can be useful if we are going to gather the statistics for a group of servers working in parallel, although the best practice would be to use the `serverProcessor` function only once per each server.

21 Example: Queue Network

To illustrate how the streams and processors can be used for modeling, let us consider a model [16, 26] of two work stations connected in a series and separated by finite queues.

> The maintenance facility of a large manufacturer performs two operations. These operations must be performed in series; operation 2 always follows operation 1. The units that are maintained are bulky, and space is available for only eight units including the units being worked on. A proposed design leaves space for two units between the work stations, and space for four units before work station 1. [..] Current company policy is to subcontract the maintenance of a unit if it cannot gain access to the in-house facility.
>
> Historical data indicates that the time interval between requests for maintenance is exponentially distributed with a mean of 0.4 time units. Service times are also exponentially distributed with the first

station requiring on the average 0.25 time units and the second station, 0.5 time units. Units are transported automatically from work station 1 to work station 2 in a negligible amount of time. If the queue of work station 2 is full, that is, if there are two units awaiting for work station 2, the first station is blocked and a unit cannot leave the station. A blocked work station cannot server other units.

Below is provided a more general model, where some work stations can serve in parallel or sequentially. The goal is to represent a simulation framework that can be used for modeling significantly more complex behaviors.

Also it is worth noting that here we do not use the *proc*-notation of Haskell. It is enough to have the standard arrow combinator (>>>) to connect the processors together.

```
import Prelude hiding (id, (.))

import Control.Monad
import Control.Monad.Trans
import Control.Arrow
import Control.Category (id, (.))

import Simulation.Aivika
import Simulation.Aivika.Queue

-- the mean delay of the input arrivals distributed exponentially
meanOrderDelay = 0.4

-- the capacity of the queue before the first work places
queueMaxCount1 = 4

-- the capacity of the queue before the second work places
queueMaxCount2 = 2

-- the mean processing time distributed exponentially in
-- the first work stations
meanProcessingTime1 = 0.25

-- the mean processing time distributed exponentially in
-- the second work stations
meanProcessingTime2 = 0.5

-- the number of the first work stations
-- (in parallel but the commented code allocates them sequentially)
workStationCount1 = 1

-- the number of the second work stations
-- (in parallel but the commented code allocates them sequentially)
workStationCount2 = 1

-- create a work station (server) with the exponential processing time
newWorkStationExponential meanTime =
  newServer $ \a ->
  do holdProcess =<<
       (liftParameter $
        randomExponential meanTime)
     return a

model :: Simulation ...
model = do
```

```
-- define a stream of input events
let inputStream = randomExponentialStream meanOrderDelay

-- create a queue before the first work stations
queue1 <-
  runEventInStartTime $
  newFCFSQueue queueMaxCount1

-- create a queue before the second work stations
queue2 <-
  runEventInStartTime $
  newFCFSQueue queueMaxCount2

-- create the first work stations (servers)
workStation1s <- forM [1 .. workStationCount1] $ \_ ->
  newWorkStationExponential meanProcessingTime1

-- create the second work stations (servers)
workStation2s <- forM [1 .. workStationCount2] $ \_ ->
  newWorkStationExponential meanProcessingTime2

-- processor for the queue before the first work station
let queueProcessor1 =
      queueProcessor
      (\a -> liftEvent $ enqueueOrLost_ queue1 a)
      (dequeue queue1)

-- processor for the queue before the second work station
let queueProcessor2 =
      queueProcessor
      (enqueue queue2)
      (dequeue queue2)

-- the entire processor from input to output
let entireProcessor =
      queueProcessor1 >>>
      processorParallel (map serverProcessor workStation1s) >>>
      -- processorSeq (map serverProcessor workStation1s) >>>
      queueProcessor2 >>>
      processorParallel (map serverProcessor workStation2s)
      -- processorSeq (map serverProcessor workStation2s)

-- start simulating the model
runProcessInStartTime $
  sinkStream $ runProcessor entireProcessor inputStream

...
```

The end part shows who we should run the queue network. We have to read permanently data from the output stream generated by the second work station. It initiates the process of receiving data from the queue located in a space between the both work stations. The corresponding queue processor begins requesting the first work station, which in its turn initiates the process of receiving data from the first queue, which processor begins reading data from the input random stream of data distributed exponentially.

Although the model is complete per se, the example itself is not yet complete. We should also save and probably analyze the simulation results. The corresponded part is hidden by dots. The real simulation model preparing the results in a short text form is included in the Aivika distribution. But you will probably be more interested in that how the results could be automatically plotted as charts, saved in files and probably analyzed. There is the corresponded support provided by additional packages complementing the Aivika library. See section 26 for more details.

22 Agent-based Modeling

Aivika provides a basic support of the agent-based modeling[25].

An idea is to try to describe a model as a cooperative behavior of a relatively large number of small agents. The agents can have states and these states can be either active or inactive. We can assign to the state a handler that is actuated under the condition that the state remains active.

We create new agents and their states within the `Simulation` computation.

```
data Agent
data AgentState

newAgent :: Simulation Agent
newState :: Agent -> Simulation AgentState
newSubstate :: AgentState -> Simulation AgentState
```

Only one of the states can be selected for each agent at the modeling time. All ancestor states remain active if they were active before, or they become active if they were deactivated. Other states are deactivated if they were active on the contrary.

```
selectedState :: Agent -> Event (Maybe AgentState)
selectState :: AgentState -> Event ()
```

The first function returns the currently selected state or `Nothing` if the agent was not yet initiated. Other function allows selecting a new state. The both functions return actions within the `Event` computation, which means that the state selection is always synchronized with the event queue.

We can assign the `Event` handlers to be performed when activating or deactivating the specified third state during such a selection.

```
setStateActivation :: AgentState -> Event () -> Event ()
setStateDeactivation :: AgentState -> Event () -> Event ()
```

If the specified third state remains active when selecting another state, but the path from the old selected state to a new state goes through the third state, then we can call the third state transitive and can assign an action to be performed when such a transition occurs.

```
setStateTransition :: AgentState -> Event (Maybe AgentState) -> Event ()
```

Here the new selected state is sent to the corresponded `Event` computation.

What differs the agents from other simulation concepts is an ability to assign so called *timeout* and *timer handlers*. The timeout handler is an `Event` computation which is actuated in the specified time interval if the sate remains active. The timer handler is similar, but only the handler is repeated while the state still remains active. Therefore, the timeout handler accepts the time as a pure value, while the timer handler recalculates the time interval within the `Event` computation after each successful actualization.

```
addTimeout :: AgentState -> Double -> Event () -> Event ()
addTimer :: AgentState -> Event Double -> Event () -> Event ()
```

The implementation is quite simple. By the specified state handler, we create a wrapper handler which we pass in to the `enqueueEvent` function with the desired time of actuating. If the state becomes deactivated before the planned time comes then we invalidate the wrapper. After the wrapper is actuated by the event queue at the planned time, we do not call the corresponded state handler if the wrapper was invalidated earlier.

We use the `Event` computation to synchronize the agents with the event queue. It literally means that the agent-based modeling can be integrated with other simulation methods within one combined model.

23 Example: Agent-based Modeling

To illustrate the use of agents, let us take the Bass Diffusion model from the AnyLogic documentation [25].

> The model describes a product diffusion process. Potential adopters of a product are influenced into buying the product by advertising and by word of mouth from adopters ? those who have already purchased the new product. Adoption of a new product driven by word of mouth is likewise an epidemic. Potential adopters come into contact with adopters through social interactions. A fraction of these contacts results in the purchase of the new product. The advertising causes a constant fraction of the potential adopter population to adopt each time period.

Below is provided a complete Haskell program. It runs the simulation and just prints the number of potential adopters in the "integration" time points[5]. Here we could complicate the output part and plot the deviation chart to provide the sensitivity analysis with help of simulation experiments described in section 26.

```
import System.Random
import Data.Array
import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika

n = 500    -- the number of agents

advertisingEffectiveness = 0.011
contactRate = 100.0
adoptionFraction = 0.015

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 8.0,
                spcDT = 0.1,
                spcMethod = RungeKutta4,
```

[5]The integration time points are specified by simulation specs. They are often used for numeric integration, but here we can use them for printing the reference values with time increment. Actually, here word "integration" has a more convenient sense than real.

```
                spcGeneratorType = SimpleGenerator }

data Person = Person { personAgent :: Agent,
                       personPotentialAdopter :: AgentState,
                       personAdopter :: AgentState }

createPerson :: Simulation Person
createPerson =
  do agent <- newAgent
     potentialAdopter <- newState agent
     adopter <- newState agent
     return Person { personAgent = agent,
                     personPotentialAdopter = potentialAdopter,
                     personAdopter = adopter }

createPersons :: Simulation (Array Int Person)
createPersons =
  do list <- forM [1 .. n] $ \i ->
       do p <- createPerson
          return (i, p)
     return $ array (1, n) list

definePerson :: Person -> Array Int Person -> Ref Int -> Ref Int -> Event ()
definePerson p ps potentialAdopters adopters =
  do setStateActivation (personPotentialAdopter p) $
       do modifyRef potentialAdopters $ \a -> a + 1
          -- add a timeout
          t <- liftParameter $
               randomExponential (1 / advertisingEffectiveness)
          let st  = personPotentialAdopter p
              st' = personAdopter p
          addTimeout st t $ selectState st'
     setStateActivation (personAdopter p) $
       do modifyRef adopters  $ \a -> a + 1
          -- add a timer that works while the state is active
          let t = liftParameter $
                  randomExponential (1 / contactRate)    -- many times!
          addTimer (personAdopter p) t $
            do i <- liftIO $ getStdRandom $ randomR (1, n)
               let p' = ps ! i
               st <- selectedState (personAgent p')
               when (st == Just (personPotentialAdopter p')) $
                 do b <- liftParameter $
                         randomTrue adoptionFraction
                    when b $ selectState (personAdopter p')
     setStateDeactivation (personPotentialAdopter p) $
       modifyRef potentialAdopters $ \a -> a - 1
     setStateDeactivation (personAdopter p) $
       modifyRef adopters $ \a -> a - 1

definePersons :: Array Int Person -> Ref Int -> Ref Int -> Event ()
definePersons ps potentialAdopters adopters =
  forM_ (elems ps) $ \p ->
  definePerson p ps potentialAdopters adopters

activatePerson :: Person -> Event ()
activatePerson p = selectState (personPotentialAdopter p)

activatePersons :: Array Int Person -> Event ()
activatePersons ps =
  forM_ (elems ps) $ \p -> activatePerson p
```

```
model :: Simulation [IO [Int]]
model =
  do potentialAdopters <- newRef 0
     adopters <- newRef 0
     ps <- createPersons
     runEventInStartTime $
       do definePersons ps potentialAdopters adopters
          activatePersons ps
     runDynamicsInIntegTimes $
       runEvent $
       do i1 <- readRef potentialAdopters
          i2 <- readRef adopters
          return [i1, i2]

main =
  do xs <- runSimulation model specs
     forM_ xs $ \x -> x >>= print
```

The reader can notice that the model uses the same monads that we used for the ordinary differential equations and discrete event simulation.

24 Circuit

In the recent versions of Aivika the author added a new computation that allows modeling a circuit with recursive links and delays. It is inspired by an idea of automata described in the literature[14, 7], which is consonant to the approach applied in Yampa.

```
newtype Circuit a b = Circuit { runCircuit :: a -> Event (b, Circuit a b) }
```

This is an automaton that takes an input and returns the next state and output within the `Event` computation synchronized with the event queue.

The `Circuit` type is obviously an `ArrowLoop`. Therefore, we can create recursive links and introduce delays by one step using the specified initial value.

```
delayCircuit :: a -> Circuit a a
```

Also we can integrate numerically the differential equations within the circuit computation creating loopbacks using the *proc*-notation if required.

```
integCircuit :: Double  -> Circuit Double Double
```

By the specified initial value we return a circuit that treats the input as derivative and returns the integral value as output.

In a similar way we can solve numerically a system of difference equations, where the next function takes the initial value too but returns an automaton that generates the sum as output by the input specifying the difference.

```
sumCircuit :: Num a => a -> Circuit a a
```

So, the system of ODEs from section 4 can be rewritten as follows.

```
circuit :: Circuit () [Double]
circuit =
  let ka = 1
      kb = 1
  in proc () -> do
```

```
    rec a  <- integCircuit 100 -< - ka * a
        b  <- integCircuit 0 -< ka * a - kb * b
        c  <- integCircuit 0 -< kb * b
    returnA -< [a, b, c]
```

To get the final results of integration, now we have to transform somehow the `Circuit` arrow computation. Therefore we need some conversion.

An arbitrary circuit can be treated as the signal transform or processor.

```
circuitSignaling :: Circuit a b -> Signal a -> Signal b
circuitProcessor :: Circuit a b -> Processor a b
```

Furthermore, the circuit can be approximated in the integration time points and interpolated in other time points:

```
circuitTransform :: Circuit a b -> Transform a b
```

Here the `Transform` type is similar to the ending part of the `integ` function. It can be realized as an analogous circuit as opposite to digital one represented by the `Circuit` computation.

```
newtype Transform a b =
  Transform { runTransform :: Dynamics a -> Simulation (Dynamics b) }
```

Also it gives us another interpretation for the `Dynamics` computation. The latter can be considered as a single entity defined in all time points simultaneously, which seems to be natural as we can approximate the integral in such a way. Consider comparing this computation with the `Event` computation, where we strongly emphasize on the fact that the `Event` computation is bound up with the current simulation time point. Speaking of the `Dynamics` computation, we do not do any assumptions regarding the simulation time.

By the way, the `Transform` type is `ArrowLoop` but not `ArrowChoice` unlike `Circuit`, which also seems to be reasonable as the `Dynamics` computation usually approximates *continuous* functions, while the `Event` computation is *discrete* by all its nature despite of the fact that both computations have absolutely the same internal representation but differ on level of the Haskell type system.

Returning to our ODE, we can run the model approximating the circuit in the integration time points for simplicity, but not for efficiency, though.

```
model :: Simulation [Double]
model =
  do results <-
       runTransform (circuitTransform circuit) $
       return ()
     runDynamicsInStopTime results
```

This model will return almost the same results[6] but it is much slower than the model that used the `integ` function within the `Dynamics` computation.

The `integCircuit` function itself has a very small memory footprint, or more precisely, it creates a lot of small short-term functions but recycles them immediately. But this footprint is mostly neglected by the `circuitTransform` function, which is memory consuming.

[6]Even if we will use Euler's method which must be equivalent, there is also an inevitable inaccuracy of calculations.

On the contrary, the `integ` function may allocate a potentially huge array once, but then it consumes almost no memory when integrating numerically.

Comparing the circuit with other computations, the former always returns its output at the current modeling time without delay, but the circuit also saves its state and we can request for the next output by next input at any desired time later. It is essential that the circuit allows specifying recursive links, which can be useful to describe a simple digital circuit, for example.

The signal transform works in another way. Every signal is an actor in the sense that it is triggered within its underlying computation. We either receive a signal or lose it. After we create the signal transform, the output signal can be triggered in time points different from the time points at which the source signal was received. There can be delays and even some signals can be omitted or filtered.

The processor is quite different. It already transforms the specified stream of data and returns a new stream. Each stream is passive in the sense that something outside must request for the next data and receive a tail of the stream to proceed later. Moreover, the stream can suspend the simulation process and return an output already at a later time point. Unfortunately, the processor cannot be defined using recursive links as it is based on continuations. Nevertheless, the processor and streams can suit very well to higher level modeling of queue networks.

An approximation of the circuit in the integration time points allows using it in the differential equations. A synchronization with the event queue is provided automatically.

25 Network

There is a version of the `Circuit` computation but only with the `Event` computation replaced by the `Process` computation.

```
newtype Net a b = Net { runNet :: a -> Process (b, Net a b) }
```

The `Net` type has a more efficient implementation of the `Arrow` type class than `Processor`. A computation received with help of the *proc*-notation must be significantly more light-weight. It is similar to the `Circuit` computation, but only `Net` is not `ArrowLoop`, because the `Process` monad is not `MonadFix`, being based on continuations.

Moreover, the `Net` computation can be easily converted to a processor and it can be done very efficiently, which shows the main use case for this type: writing some parts of the model within `Net` computation using the *proc*-notation with the further conversion.

```
netProcessor :: Net a b -> Processor a b
netProcessor = Processor . loop
  where loop x as =
          Cons $
          do (a, as') <- runStream as
             (b, x') <- runNet x a
             return (b, loop x' as')
```

The problem is that the `Net` type has no clear multiplexing and demultiplexing facilities for parallelizing the processing. Also its opposite conversion is quite costly and there is actually no guarantee that the specified processor will produce exactly one output for each input value.

```
processorNet :: Processor a b -> Net a b
```

Nevertheless, the both computations can be very useful in their combination.

26 Simulation Experiment

Aivika extensively uses the monads and the final results of simulation must be returned within the `Simulation` computation.

In theory, we can return almost any data structure we might desire. But it seems to be quite tedious and routine. Moreover, the results should be represented in a form suitable for analysis.

Therefore, the author created additional libraries Aivika Experiment[22] and Aivika Experiment Chart[20] with different rendering backends[19, 21]. The packages allow defining and then running the simulation experiments, where we can in a relatively short and declarative way specify what data we want to return and how these data must be represented.

Since Aivika of version 1.4, a module of exporting the simulation results was added to the main package.

For example, our ODE can return the `Results` data value that encompasses the variable values and their names.

```
model :: Simulation Results
model =
  mdo a <- integ (- ka * a) 100
      b <- integ (ka * a - kb * b) 0
      c <- integ (kb * b) 0
      let ka = 1
          kb = 1
      return $ results
        [resultSource "a" "variable A" a,
         resultSource "b" "variable B" b,
         resultSource "c" "variable C" c]
```

The key function is `resultSource` that creates a source of the simulation results. Here we passed in ordinary `Dynamics` computations, but the function, being defined in a type class, can understand arrays, lists as well as other simulation computations and even the `Ref` references, `Var` variables, queues, servers and so on.

We can run the model and print the simulation results in a text form, for example.

```
main =
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

Then we will see the following output in Terminal:

```
----------

-- simulation time
t = 13.0
```

```
-- variable A
a = 2.260329409450236e-4

-- variable B
b = 2.938428231048658e-3

-- variable C
c = 99.99683553882805
```

It is more interesting if we specify more complex structures such as queues that have a lot of different properties. The library will print these properties.

In case of need we can post-process the results, for example, shortening the number of properties as some of them are important than others, which is especially useful for the same queues.

```
data Results
type ResultTransform = Results -> Results

resultSummary :: ResultTransform
```

The text output is the first stage only. We can go further with the mentioned packages Aivika Experiment and Aivika Experiment Chart.

When running, the simulation experiment can generate an HTML page as well as a plenty of charts and tables we want to receive. The HTML page and charts can be observed in the Internet browser, while the tables can be loaded into the Office or statistical application for the further processing.

At present these libraries can plot the time series, the XY chart, the deviation chart (the 3-sigma rule) and histograms. The simulation results in a form of table can be saved in the CSV files. Moreover, the libraries support the Monte-Carlo simulation. In perspective, optimization facilities can be added as an additional library too.

All this makes the Aivika library a practical tool for modeling, simulation and analysis of the results. Aivika has both a theoretical and practical values in the author's opinion.

27 Example: Parametric Model

Now we will focus on a practical question: how to prepare a parametric model for the Monte-Carlo simulation? For example, it can be useful for providing the sensitivity analysis.

Let us take the financial model[27] described in Vensim 5 Modeling Guide, Chapter Financial Modeling and Risk. Probably, the best way to describe the model is just to show its equations.

The equations use the `npv` function from System Dynamics. It returns the Net Present Value (NPV) of the stream computed using the specified discount rate, the initial value and some factor (usually 1).

```
npv :: Dynamics Double                  -- ^ the stream
       -> Dynamics Double               -- ^ the discount rate
       -> Dynamics Double               -- ^ the initial value
       -> Dynamics Double               -- ^ factor
       -> Simulation (Dynamics Double)  -- ^ the Net Present Value (NPV)
npv stream rate init factor =
  mdo let dt' = liftParameter dt
      df <- integ (- df * rate) 1
      accum <- integ (stream * df) init
      return $ (accum + dt' * stream * df) * factor
```

Also we need a helper conditional combinator that allows avoiding the *do*-notation in some cases.

```
-- | Implement the if-then-else operator.
ifDynamics :: Dynamics Bool -> Dynamics a -> Dynamics a -> Dynamics a
ifDynamics cond x y =
  do a <- cond
     if a then x else y
```

After we finished the necessary preliminaries, now we can show how the parametric model can be prepared for the Monte-Carlo simulation.

We represent each external parameter as a `Parameter` computation. To be reproducible within every simulation run, the random parameter must be memoized with help of the `memoParameter` function.

Also this model returns the `Results` within the `Simulation` computation. Such results can be processed then or just printed in Terminal.

```
{-# LANGUAGE RecursiveDo #-}

module Model
       (-- * Simulation Model
        model,
        -- * Variable Names
        netIncomeName,
        netCashFlowName,
        npvIncomeName,
        npvCashFlowName,
        -- * External Parameters
        Parameters(..),
        defaultParams,
        randomParams) where

import Control.Monad

import Simulation.Aivika
import Simulation.Aivika.SystemDynamics

-- | The model parameters.
data Parameters =
  Parameters { paramsTaxDepreciationTime    :: Parameter Double,
               paramsTaxRate                :: Parameter Double,
               paramsAveragePayableDelay    :: Parameter Double,
               paramsBillingProcessingTime  :: Parameter Double,
               paramsBuildingTime           :: Parameter Double,
               paramsDebtFinancingFraction  :: Parameter Double,
               paramsDebtRetirementTime     :: Parameter Double,
               paramsDiscountRate           :: Parameter Double,
               paramsFractionalLossRate     :: Parameter Double,
               paramsInterestRate           :: Parameter Double,
               paramsPrice                  :: Parameter Double,
               paramsProductionCapacity     :: Parameter Double,
               paramsRequiredInvestment     :: Parameter Double,
               paramsVariableProductionCost :: Parameter Double }

-- | The default model parameters.
```

```haskell
defaultParams :: Parameters
defaultParams =
  Parameters { paramsTaxDepreciationTime    = 10,
               paramsTaxRate                = 0.4,
               paramsAveragePayableDelay    = 0.09,
               paramsBillingProcessingTime  = 0.04,
               paramsBuildingTime           = 1,
               paramsDebtFinancingFraction  = 0.6,
               paramsDebtRetirementTime     = 3,
               paramsDiscountRate           = 0.12,
               paramsFractionalLossRate     = 0.06,
               paramsInterestRate           = 0.12,
               paramsPrice                  = 1,
               paramsProductionCapacity     = 2400,
               paramsRequiredInvestment     = 2000,
               paramsVariableProductionCost = 0.6 }

-- | Random parameters for the Monte-Carlo simulation.
randomParams :: IO Parameters
randomParams =
  do averagePayableDelay    <- memoParameter $ randomUniform 0.07 0.11
     billingProcessingTime  <- memoParameter $ randomUniform 0.03 0.05
     buildingTime           <- memoParameter $ randomUniform 0.8 1.2
     fractionalLossRate     <- memoParameter $ randomUniform 0.05 0.08
     interestRate           <- memoParameter $ randomUniform 0.09 0.15
     price                  <- memoParameter $ randomUniform 0.9 1.2
     productionCapacity     <- memoParameter $ randomUniform 2200 2600
     requiredInvestment     <- memoParameter $ randomUniform 1800 2200
     variableProductionCost <- memoParameter $ randomUniform 0.5 0.7
     return defaultParams { paramsAveragePayableDelay    = averagePayableDelay,
                            paramsBillingProcessingTime  = billingProcessingTime,
                            paramsBuildingTime           = buildingTime,
                            paramsFractionalLossRate     = fractionalLossRate,
                            paramsInterestRate           = interestRate,
                            paramsPrice                  = price,
                            paramsProductionCapacity     = productionCapacity,
                            paramsRequiredInvestment     = requiredInvestment,
                            paramsVariableProductionCost =
                                variableProductionCost }

-- | This is the model itself that returns experimental data.
model :: Parameters -> Simulation Results
model params =
  mdo let getParameter f = liftParameter $ f params

      -- the equations below are given in an arbitrary order!

      bookValue <- integ (newInvestment - taxDepreciation) 0
      let taxDepreciation = bookValue / taxDepreciationTime
          taxableIncome = grossIncome - directCosts - losses
                          - interestPayments - taxDepreciation
          production = availableCapacity
          availableCapacity = ifDynamics (time .>=. buildingTime)
                              productionCapacity 0
          taxDepreciationTime = getParameter paramsTaxDepreciationTime
          taxRate = getParameter paramsTaxRate
      accountsReceivable <- integ (billings - cashReceipts - losses)
                            (billings / (1 / averagePayableDelay
                                         + fractionalLossRate))
      let averagePayableDelay = getParameter paramsAveragePayableDelay
      awaitingBilling <- integ (price * production - billings)
                         (price * production * billingProcessingTime)
      let billingProcessingTime = getParameter paramsBillingProcessingTime
          billings = awaitingBilling / billingProcessingTime
          borrowing = newInvestment * debtFinancingFraction
          buildingTime = getParameter paramsBuildingTime
          cashReceipts = accountsReceivable / averagePayableDelay
      debt <- integ (borrowing - principalRepayment) 0
      let debtFinancingFraction = getParameter paramsDebtFinancingFraction
          debtRetirementTime = getParameter paramsDebtRetirementTime
          directCosts = production * variableProductionCost
          discountRate = getParameter paramsDiscountRate
          fractionalLossRate = getParameter paramsFractionalLossRate
          grossIncome = billings
          interestPayments = debt * interestRate
          interestRate = getParameter paramsInterestRate
          losses = accountsReceivable * fractionalLossRate
          netCashFlow = cashReceipts + borrowing - newInvestment
                        - directCosts - interestPayments
                        - principalRepayment - taxes
          netIncome = taxableIncome - taxes
          newInvestment = ifDynamics (time .>=. buildingTime)
                          0 (requiredInvestment / buildingTime)
      npvCashFlow <- npv netCashFlow discountRate 0 1
      npvIncome <- npv netIncome discountRate 0 1
      let price = getParameter paramsPrice
          principalRepayment = debt / debtRetirementTime
          productionCapacity = getParameter paramsProductionCapacity
          requiredInvestment = getParameter paramsRequiredInvestment
          taxes = taxableIncome * taxRate
          variableProductionCost = getParameter paramsVariableProductionCost

      return $
        results
        [resultSource netIncomeName "Net income" netIncome,
         resultSource netCashFlowName "Net cash flow" netCashFlow,
         resultSource npvIncomeName "NPV income" npvIncome,
         resultSource npvCashFlowName "NPV cash flow" npvCashFlow]

-- the names of the variables we are interested in
netIncomeName   = "netIncome"
netCashFlowName = "netCashFlow"
npvIncomeName   = "npvIncome"
npvCashFlowName = "npvCashFlow"
```

Now we can apply the Monte-Carlo simulation to this parametric model, for example, to define how sensitive are some variables to the random external parameters.

The point is that not only ODEs can be parametric. There is not any difference, whether we integrate numerically, or run the discrete event simulation, or simulate the agents. The external parameters are just `Parameter` computations that can be used within other simulation computations.

Returning to the example, the reader can find a complete version of the model in the Aivika Experiment Chart package.

28 Example: Using Arrays

Some vendors offer different versions of their simulation software tools, where one of the main advantages of using a commercial version is an ability to use arrays.

There is no need in special support of arrays in Aivika. They can be naturally used with the simulation computations.

Let us take model Linear Array from Berkeley Madonna[9] to demonstrate the main idea.

```
{-# LANGUAGE RecursiveDo #-}

module Model (model) where

import Data.Array
import Control.Monad
import Control.Monad.Trans

import qualified Data.Vector as V

import Simulation.Aivika
import Simulation.Aivika.SystemDynamics

-- | This is an analog of 'V.generateM' included in the Haskell platform.
generateArray :: (Ix i, Monad m) => (i, i) -> (i -> m a) -> m (Array i a)
generateArray bnds generator =
  do ps <- forM (range bnds) $ \i ->
       do x <- generator i
          return (i, x)
     return $ array bnds ps

model :: Int -> Simulation Results
model n =
  mdo m <- generateArray (1, n) $ \i ->
        integ (q + k * (c!(i - 1) - c!i) + k * (c!(i + 1) - c!i)) 0
      let c =
            array (0, n + 1) [(i, if (i == 0) || (i == n + 1)
                                  then 0
                                  else (m!i / v)) | i <- [0 .. n + 1]]
          q = 1
          k = 2
          v = 0.75
      return $ results
        [resultSource "t" "time" time,
         resultSource "m" "M" m,
         resultSource "c" "C" c]
```

The code provided above uses the standard `Array` module. If we used a new `Vector` module, then we would need no function `generateArray` at all.

This model creates an array of integrals. Similarly, we could use arrays in the discrete event simulation or agent-based model.

29 Related Work

Quite a different approach for integrating the ODE system as well as for modeling the discrete event simulation is implemented in Yampa, a Haskell library mentioned before, primarily basing on the FRP approach and using the arrow combinators. In Yampa the integral is defined in terms of the arrow but the events form an infinite stream.

In the recent versions of Aivika the author adapted some ideas from Yampa. The result is the new `Circuit` computation.

Another known Haskell simulation library Hydra introduces a Functional Hybrid Modeling (FHM) language for modeling and simulation of physical systems using implicitly formulated (undirected) Differential Algebraic Equations (DAEs). The approach is essentially based on using quasi-quoting to simulate such systems.

On the contrary, Aivika does not use quasi-quoting. Moreover, the system of ODEs is defined within the monadic computation, although some quasi-quoting could be used in the future for speeding up the numerical integration as well as for formulating DAEs and the example of Hydra could be very useful here.

Regarding the continuations, which the Aivika approach for discrete event simulation is essentially based on, they are used in Lisp for a long time. The Racket package Simulation Collection[28] extensively uses the continuations for the process-oriented simulation. The library also allows building continuous simulation models.

There is also another approach used for modeling the discontinuous processes. The SimPy[24] library written in the Python programming language uses generators to implement the process-oriented simulation. The generators are close to the `yield return` construct from the C# programming language.

Finally, master's thesis[2] by Nikolaos Bezirgiannis is devoted to improving the performance of simulation basing on Haskell's concurrency and parallelism.

30 Conclusions

The suggested simulation formalism is based on the known ideas approved in the practice. The author mainly compiled these ideas from different sources and applied them to the simulation field. Basing on this formalism, he developed open-source software tools that simplify modeling, simulation and analysis of the results.

The library is designed in such a way that the specialized computations are destined for modeling the specialized tasks.

For example, the integral is well described by one computation, an event-based activity should be described by another computation, the process can be modeled with help of continuations, while a high-level queue network can be described in terms on infinite streams and their transformations, and all this can be combined within one hybrid model.

The very idea of an abstract computation, being it a monad, stream or an arrow, is very natural for modeling the simulation activity. What is important, Haskell provides the corresponded syntax sugar, i.e. the *do*-notation, the recursive *do*-notation and the *proc*-notation, which is crucial for the method's applicability.

Acknowledgments

The author is grateful to Eugene Kirphichov for helpful comments and discussions. Many thanks to Christophe Favergeon for his suggestions and feedback. Also thanks to Mike Izbicki for the consent to use his code for creating histograms.

Finally, I very appreciate the recommendations of my anonymous readers who pointed me to weak places in the draft of this document. I tried to listen to them and make the paper better.

References

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., USA, 1985.

[2] Nicolaos Bezirgiannis. Improving performance of simulation software using Haskell's concurrency and parallelism. Master's thesis, Dept. of Information and Computing Sciences, Utrecht University, 2013.

[3] George Giorgidze. `Hydra` Library. `https://github.com/giorgidze/Hydra`, 2012. Accessed: 28-June-2014.

[4] George Giorgidze and Henrik Nilsson. Embedding a functional hybrid modelling language in haskell. In *In Refereed Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL '08), University of Hertfordshire*, 2008.

[5] Paul Hudak. *The Haskell School of Expression.* Cambridge University Press, 2007.

[6] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

[7] John Hughes. Programming with arrows. In *Advanced Functional Programming*, pages 73–129, 2004.

[8] `iThink` Software. `http://www.iseesystems.com`, 2014. Accessed: 1-May-2014.

[9] Robert Macey and George Oster. `Berkeley Madonna` Software. `http://www.berkeleymadonna.com`, 2014. Accessed: 1-May-2014.

[10] Norm Matloff. Introduction to discrete-event simulation and the `SimPy` language. `http://simpy.readthedocs.org/en/latest/`, 2008. Accessed: 1-May-2014.

[11] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 54–65, New York, NY, USA, 2005. ACM.

[12] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[13] Henrik Nilsson and Antony Courtney et al. `Yampa` Library, Version 0.9.5. `http://hackage.haskell.org/package/Yampa`, 2014. Accessed: 1-May-2014.

[14] Ross Paterson. A new notation for Arrows. In *In International Conference on Functional Programming*, ICFP '01, pages 229–240. ACM, 2001.

[15] Tomas Petricek and Jon Skeet. Programming user interfaces using F# workflows. `http://dotnetslackers.com/articles/net/Programming-user-interfaces-using-f-sharp-workflows.aspx`, 2010. Accessed: 1-May-2014.

[16] A.A.B. Pritsker and J.J. O'Reilly. *Simulation with Visual SLAM and AweSim*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1999.

[17] David E. Sorokin. `Scala Aivika` Library, Version 1.0. `https://github.com/dsorokin/scala-aivika`, 2012. Accessed: 1-May-2014.

[18] David E. Sorokin. `Aivika` F# Library, Version 2.0.14.0. `http://sourceforge.net/projects/aivika/`, 2013. Accessed: 1-May-2014.

[19] David E. Sorokin. `Aivika Experiment Cairo` Library, Version 1.3. `http://hackage.haskell.org/package/aivika-experiment-cairo`, 2014.

[20] David E. Sorokin. `Aivika Experiment Chart` Library, Version 1.3. `http://hackage.haskell.org/package/aivika-experiment-chart`, 2014. Accessed: 28-June-2014.

[21] David E. Sorokin. `Aivika Experiment Diagrams` Library, Version 1.3. `http://hackage.haskell.org/package/aivika-experiment-diagrams`, 2014.

[22] David E. Sorokin. `Aivika Experiment` Library, Version 1.3. `http://hackage.haskell.org/package/aivika-experiment`, 2014. Accessed: 28-June-2014.

[23] David E. Sorokin. `Aivika` Library, Version 1.3. `http://hackage.haskell.org/package/aivika`, 2014. Accessed: 28-June-2014.

[24] `SimPy` Library. `http://simpy.readthedocs.org/en/latest/`, 2014. Accessed: 1-May-2014.

[25] `AnyLogic` Software. `http://www.anylogic.com`, 2014. Accessed: 1-May-2014.

[26] Ilya I. Trub. *An Object-oriented Modeling in C++*. Piter, Russia, 2006. (In Russian).

[27] `Vensim` Software. `http://vensim.com`, 2013. Accessed: 1-May-2014.

[28] M. Douglas Williams. `Simulation Collection` Library, Version 3.5. `http://planet.racket-lang.org/display.ss?package=simulation.plt&owner=williams`, 2012. Accessed: 1-May-2014.