Modeling Genome Evolution with a DSEL for Probabilistic Programming

Martin Erwig and Steve Kollmansberger

School of EECS
Oregon State University
[erwig|kollmast]@eecs.oregonstate.edu

Abstract. Many scientific applications benefit from simulation. However, programming languages used in simulation, such as C++ or Matlab, approach problems from a deterministic procedural view, which seems to differ, in general, from many scientists' mental representation. We apply a domain-specific language for probabilistic programming to the biological field of gene modeling, showing how the mental-model gap may be bridged. Our system assisted biologists in developing a model for genome evolution by separating the concerns of model and simulation and providing implicit probabilistic non-determinism.

Keywords: Functional Programming, Probabilistic Programming, Haskell, Genome Evolution

1 Introduction

A primary occupation of scientists is to devise models of observable processes. These models may be formal and mathematical, or informal ideas and sketches. In general, such models cannot be executed, simulated nor verified directly. Instead, scientists have had to translate their model first into a programming language. Traditionally, simulations for scientific models were written in the programming language of their day, such as Fortran or C. Later simulations were also written in mathematical packages such at MatLab. Recently, some researchers have developed domain-specific modeling tools for biological processes [9, 11, 15, 4, 3].

Many of these approaches, however, are merely speculation and have not been used in an actual research application. In addition, many of them are limited to only the particular given model, and so general computation cannot be mixed with the scientific specification. For example, the bio-ambients approach requires any model to be given in terms of a hierarchical chain of interacting objects [15]. On the other hand, some approaches are too general, forcing scientists to adapt their ideas to fit the general-purpose constructs given by the system. For example, the pathway logic system presents a general algebraic rewrite approach without specific support for constructs that may appear in biological systems [4].

We approach the problem driven by a specific application: In conjunction

with the Center for Gene Research at Oregon State University, we have developed a model for the evolution of microRNAs [2, 1], which has enabled scientists to predict what types of genome sequences are most likely to exhibit active microRNAs. This result is important since microRNAs are an essential regulatory mechanism for controlling gene expressiveness. The model is realized with the help of a domain-specific embedded language (DSEL) for probabilistic programming [5]. This paper reports on our process as well as results applicable to modeling a wide variety of scientific domains.

We have chosen a DSEL approach because it yields a language that offers constructs general enough to represent any computation, but specific enough to be very closely related to the model. We found that the scientists did not know at the outset all the precise details of the model they wanted to represent. Therefore, choosing a DSEL approach allowed rapid prototyping and iteration as we developed the model from the ground up. We constructed the DSEL in Haskell because it offered a number of unique features that allowed "behind-the-scenes" operation (through monads), allowing the written code to closely resemble the biological concepts.

The remainder of this paper is structured as follows. We introduce our approach to probabilistic functional programming in Section 2. In Section 3 we will show how this approach can be applied to a simple biological problem, the Lotka-Volterra predator-prey model. In Section 4 we will discuss the motivation, problem, and prototyping of the genome model. The final model and its scientific accomplishments will be presented in Section 5. A discussion of related work is given in Section 6. Conclusions are presented in Section 7.

2 Probabilistic Functional Programming

We have constructed a probabilistic functional programming (PFP) library [5, 13] based on a DSEL approach. The foundational structure of probabilistic computing is a list of values and their associated probabilities, called a *distribution*, which is encapsulated in the type:

```
Dist a
```

The users of the library do not directly construct such distributions—instead, we provide a variety of functions which construct and operate on them. For example, the functions `uniform` and `normal` construct a distribution from a list of values. These distributions are of course discrete, so they can be considered as approximations.

We can extract probabilities from the distribution using predicates on values in the distribution, called *events*. The function `??` takes such a predicate and determines the probability (represented by a float value) that it is true in a given distribution.

```
type Event a = a -> Bool

(??) :: Event a -> Dist a -> Probability
```

We can consider a simple example of rolling dice. A regular die has a numeric value from one to six, and may land on any of those values with equal probability.

```
type Die = Int

die :: Dist Die
die = uniform [1..6]
```

We can simulate rolling an arbitrary number of dice by the following function `dice`. The function `joinWith` combines all pairs of values from two distributions with a given function while multiplying their probabilities. In this case, we are accumulating individual die rolls in a list. The function `certainly` constructs a distribution that consists of one value with 100% probability.

```
dice :: Int -> Dist [Die]
dice 0 = certainly []
dice n = joinWith (:) die (dice (n-1))
```

Now what if we wanted to determine how likely it would be that out of a certain number of rolls, a certain number of them would come up six? Since we are producing a list, we can simply filter out all non-six values and count how long the remaining list is.

```
sixes :: (Int -> Bool) -> Int -> Probability
sixes p n = (p . length . filter (==6)) ?? dice n
```

If we wanted to determine the probability of rolling more than two sixes in a sequence of four die rolls, we could query:

```
> sixes (>2) 4
  1.6%
```

In many cases, distributions are not given directly. Instead, a series of steps are required for their construction, each one taking a value and producing a distribution. We call such a function a *transition*.

```
type Trans a = a -> Dist a
```

With transitions, distributions permit a sequenced form of computation known as a monad. In the probability monad, the function `return` indicates that a given value is certain. The bind operation `>>=` takes a distribution and a transition, threads the values in the first distribution through the transition and combines the resultant distributions. The observation that probability distributions form a monad is not new [6]. However, previous work was mainly concerned with extending languages by offering probabilistic expressions as primitives and defining suitable semantics [7, 10, 14, 12].

Consider the case where we take a sum, roll a die and add its value to the sum. We may wish to repeat this process several times. We can employ a transition which takes the current sum `s` and adds each possible die roll `d` $\in$ `die` to the sum, which is expressed using the bind operation s follows.

```
die >>= (\d->return (s+d))
```

Using Haskell's `do` notation, this expression can be rewritten in a more readable way.

```
addDie :: Trans Int
addDie s = do d <- die
              return (s+d)
```

The statement `d <- die` can be thought of as universal quantification on the values in the distribution `die`.

In many cases, we want to repeat some transition multiple times. We create a constructor class `Iterate` for repeating various kinds of transitions, represented by the type constructor `c`.

```
class Iterate c where
  (*.)  :: Int -> (a -> c a) -> (a -> c a)
  while :: (a -> Bool) -> (a -> c a) -> (a -> c a)
```

In addition to iteration over distributions, we also iterate over randomized values. These are used to avoid monotonically increasing space usage (and thus, running out of memory) that can happen iterating with full distributions. For example, consider adding the value of `n` dice. At each step, the number of possible outcomes grows. All of these distributions will be combined and threaded again through the transition. In order to avoid space expansion, we provide random selection from distributions. In randomization, a distribution is created and one value selected at random based on the probabilities. A randomized transition is called an `RChange`, which takes a value and produces one randomized value. Random numbers are computed in the `IO` monad, for which we have created the synonym `R`.

```
type RChange a = a -> R a
```

Since an `RChange` produces only one value, we can thread the value through as many steps as we want and never worry about combinatorial explosion. Randomized values may be used in monads with the same syntax as distributions, but instead of being a universal quantification, it is a single selection: an existential quantification. We also provide the ability to construct randomized distributions by repeatedly sampling a particular randomized change. A randomized distribution (`RDist`) is some collection of values and probabilities that represents an approximation of the actual distribution. This is known as a Monte Carlo sampling. An randomized transition (`RTrans`) is a function that, given a value, produces such an approximation.

```
type RDist a  = R (Dist a)
type RTrans a = a -> RDist a
```

Continuing the dice example, we can establish iteration functions for rolling and summing the value of dice. The function `dieSum` rolls one hundred dice and adds them all together. The function `rDieSum` does the same, but uses randomization to only take ten walks through the space. The function `~*.` provides randomization of a transition and repeated walks to accumulate a randomized distribution.

```
 dieSum = 100 *. addDie
rDieSum = (500,100) ~*. addDie
```

The randomized version offers considerable time and space savings. We can adjust the number of walks to spend more time gathering a better approximation, or to more quickly make a rough estimate. On the other hand, for a large number of steps, it is often impossible to run a full simulation.

The usual idea of iteration is to process a value repeatedly and return some final value. However, in some simulations we want to observe the evolution of a distribution over time. Since each step in an iteration produces an intermediate distribution, we can simply retain these distributions in a list rather than discard them. In general, a trace over any type of value can be represented as a list of that type. We call a trace of distributions a `Space`. This can be imagined as a three dimensional plane showing a slightly different distribution at each z value.

```
type Space a = [Dist a]
```

3 Probabilistic Modeling in Biology

The Lotka-Volterra predator-prey model [8] states that the population of predators and of prey can be described with mutually dependent equations. In particular, given the victims' growth factor (g), the predators' death factor (d), the search rate (s), and the energetic efficiency, (e), along with the current victim (v) and predator (p) population, a new population count can be determined with the equations $g * v - s * v * p$ (for victims) and $d * p + e * v * p$ (for predators). These new populations can then be rethreaded as input to create a simulation over time.

Consider the case when the growth and death rate are not a known constant, but exist within some probability distribution. We can define them, for example, using a normal curve.

```
growth = normal [1.01, 1.02 .. 1.10]
death  = normal [0.93, 0.94 .. 0.97]
(s,e)  = (0.01,0.01)
```

The data that we are simulating is the population of victims and predators. We can represent the population as a tuple of floats.

```
type Pop = (Float,Float)
```

Recall that we previously stated that distributions could be thought of as a monad. Monadic sequencing is very helpful in this case. We can create a transition which, given a `Pop`, produces a distribution of `Pop` based on the four distributions given above. The equations can be presented in the usual way, letting the monad do the heavy lifting of extracting values and combining probabilities.

In the transition `dvp`, all values are extracted by the monad from the distributions `growth` and `death`, and are then threaded through the equation, which is then recombined into a distribution of new values. In other words, this transition takes a current population and determines all possible new population values, and their probabilities.

```
dvp :: Trans Pop
dvp (v,p) = do g <- growth
               d <- death
               return (g*v - s*v*p `max` 0, d*p + e*v*p)
```

With an initial seed value, such as `(v0,p0) = (15,15)`, we can now simulate the predator-prey model. However, if we tried this, we would quickly find that this is a case of strong combinatorial explosive, and we would be unable to simulate more than a handful of steps! The solution is to introduce randomization. This does not require any change to our transition, nor any modification of the equation. We simply use a function to perform 1000 randomized simulations (iteration of a randomized change) to produce a randomized distribution.

```
ppt n = ((1000,n) ~.. dvp) (v0,p0)
```

Of course, having the output come as a long list of values and probabilities is neither very interesting nor very useful. Therefore, we have developed a visualization module that presents information in a graph form.

We would like to visualize the generations (steps) on the X axis and the population count on the Y axis. In order to transform a distribution of population into a single value to plot we use the `expected` function which computes the expected value of a numeric distribution.

We can devise a function which operates on a randomized space to apply the `expected` function. First, the list of distributions must be extracted from the monad, then for each element in each distribution, either the first or the second element from the tuple (representing predator or prey) must be extracted, which is done by mapping a function `f` across all elements of each distribution. The `expected` function can be applied to each distribution in the space. The application of `reverse` is needed since traces are accumulated from the most recent value to the oldest value, but we want to plot the oldest value first.

```
getRE f rs = do rs' <- rs
                let rs'' = map (fmap f) rs'
                return (reverse (map expected rs''))
```

Finally, we can produce a chart with two lines: one for the predator and one for

the prey. Note that the function `plotRL` takes a randomized list and turns it into a line on the graph. This list comes from calculating the expected value of each distribution in the randomized space.

```
fig1 = figP figure{title="Predator/Prey Simulation ",
                   xLabel="Time (generation)",
                   yLabel="Population"}
           [(plotRL v'){color=Green,label="Victim"},
            (plotRL p'){color=Red,label="Predator"}]
        where p = ppt 500
              v' = getRE fst p
              p' = getRE snd p
```

The plot created by this function is shown in Figure 1 on the left.

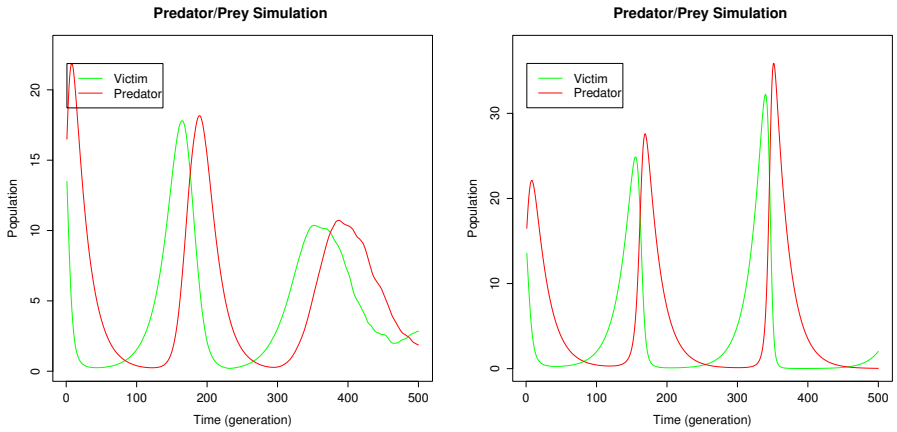

Fig. 1. Probabilistic (on the left) and deterministic (on the right) predator/prey simulation over 500 generations

Compared with the corresponding deterministic model with `growth = 1.055` and `death = 1.95` (shown in Figure 1 on the right), the probabilistic model demonstrates a quantitatively different behavior in how the peaks develop, suggesting that using probabilities in modeling has more effect than simply attempting to average the values and retain a deterministic approach. This conclusion is verified by Renshaw [16], who notes that stochastic predator-prey models almost always experience extinction after several generations.

4 Model Prototyping

In this section, we report on the gradual development of the genome model through iterations over several prototypes, followed by evaluations and discussions with biologists.

The most significant challenge we faced when developing this model was simply that the problem was not well defined; that is, the biologists did not know exactly what the model needed to represent. Thus, we have employed a method for rapid prototyping so that the model could evolve easily over time, which was essential to the project's success—the feedback and results from each step helped inform the biologists as to which direction would be most profitable to take. We conclude from our experience that any domain-specific language aimed at biologists, or scientists in general, should support rapid prototyping.

Biologists have determined that over generational time genomes experience evolutionary development. Part of this development includes parts of the genome being duplicated, and occasionally an inverted duplication. The duplications and inverted duplications can interact in some instances through microRNAs. MicroRNAs are transcribed from inverted duplications and can attach to duplicated genes to inhibit their expressiveness. In other words, when a duplication and inverted duplication are interacting, the genetic function of that duplication is suppressed. An important biological question is under what circumstances these microRNAs can develop.

To this end we had to model a genome that accumulates changes over time. The genome consists of multiple genes, which are either capable of interaction with inverted duplications or not, depending on the number of changes accumulated. The biologists felt that modeling various duplications of a single gene was sufficient. Therefore, the only information we need about each duplication (gene) is the number of changes it has accumulated. Our goal was to simulate how long any gene of the genome would remain in the state of interaction given a variety of initial conditions, such as varying rate of changes for different parts of the genome and different numbers of genes.

We started by constructing the genome as a list of duplications (also simply called genes) and inverted duplications. Duplications were simply represented as integers since the number of accumulated changes was the only information that mattered for this application. Inverted duplications had three significant parts that could accumulate changes, so we represented them with a three-tuple of integers. These three parts arise from the fact that an inverted duplication is strand of RNA folded onto itself. This can be viewed as two strands (sense and anti-sense) and a loop.

We then allowed a change to occur either in one of the parts of the inverted duplication or in one of the duplications. After discussing the model further, the biologists decided the inverted duplication needed only two components: a sense and an anti-sense. The loop was found to be non-significant. Since we were using high-level operations to express the model, the change was trivial.

Next, the biologists decided that merely having one inverted duplication was sufficient. Each duplication would then be compared against the inverted duplication to determine interaction. At this point, interaction was still a fuzzy concept, so we tried to clarify it into mathematical terms.

The biologists told us that, in the beginning, all the genes could interact with an inverted duplication. They called this state "full" interaction. Over evolution-

ary time, changes accumulate. If, for any duplication, the number of changes in that duplication plus the number of changes in the anti-sense of the inverted duplication were five or more, that duplication stopped interacting with the inverted duplication. In other words, the genetic function of that duplication could no longer be suppressed by a microRNA. If some duplications were interacting, the state was "partial". If none were interacting, the state was "none". In addition, if enough changes accumulated in the inverted duplication alone (a total of five between the sense and the anti-sense), then the inverted duplication was considered lost, and all interaction stopped. This behavior is directly implemented with the function `interaction`.

```
interaction :: Genome -> Interaction
interaction ((s,a),gs) | s+a>5 = Loss
                       | True  = if l==0 then None else
                                   if l==g then Full else Partial
                       where l=length ((filter (\n->a+n<=4)) gs)
                             g=length gs
```

It soon became apparent that this abstraction was not sufficiently detailed. The biologists told us that each gene actually needed to be divided into units. This meant that each duplication was now a list of n integers, each a place where changes could accumulate. We represented the inverted duplication as a list of n pairs (sense and anti-sense).

The additional complexity made the ideas of interaction and loss more interesting, as we had to match units in the genes with the units in the inverted duplication. We had to check each unit in a duplication against the corresponding anti-sense unit in the inverted duplication. If any had a sum of less than five changes, the duplication was still considered to interact. This concept is shown in Figure 2.

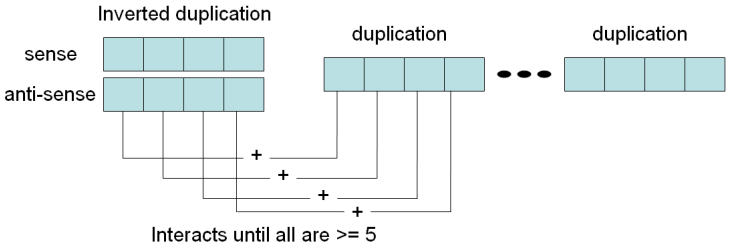

Fig. 2. The test of interaction

We made several additional changes before arriving at the final model, discussed in the next section, which the biologists found useful for generating predictions which they could test experimentally.

5 A Model of Genome Evolution

Our simulation finally ended up with a genome consisting of `Dup`s and one inverted duplication `IDup`. In addition to a given number of `Dup`s, we also had a given number of `Unit`s. Each gene was broken into that many units, and the sense and anti-sense of the inverted duplication also had that many units. We represented the inverted duplication with a list of `Bin`s, where a `Bin` is simply a pair of units.

```
type Unit = Int
type Bin  = (Unit,Unit)

type Dup    = [Unit]
type IDup   = [Bin]
type Genome = (IDup,[Dup])
```

Initially, a change could randomly occur anywhere in any unit with equal probability. However, to model evolutionary pressure, we constructed several models which defined varying degrees of resilience for the gene parts. In particular, we used a "variable model" which allowed the genes to receive all the changes that fell on them, and a "family model" which allowed only one third of the changes to the duplications to accumulate. The names "variable" and "family" derive from the biologists' labels of different classes of genes, in particular, experiments which showed that some genes were essential to the functioning of an organism (thus were resistant to change) while others could change freely. The "family model" represents those genes which are resistant to change, while the "variable model" represents those which can freely change.

A model is a function which takes the number of genes in a genome and creates a probabilistic function which selects to accumulate a change in either the genes or the inverted duplication based on the number of genes.

```
type Model = Int -> Trans Genome
```

In the function `mkModel` to create a model, `enumTT` creates a distribution of transitions. Given the number of genes, `x`, and that there are 2 parts to the inverted duplication (sense and anti-sense), we make all units equally likely to experience a change. The function `transAt` performs a transition on a pair. The parameters `2` and `1` indicate which part of the pair should have the transition applied. Since genes are the second item in the pair, the gene transition performs the identity transition on the inverted duplication and a change on the genes, while for the inverted duplication we perform a correspondingly defined change and the identity transition on the genes. The definition for `genes` considers the probability given in `gp`, representing a family ($\texttt{gp} = \frac{1}{3}$) or variable ($\texttt{gp} = 1$) model, to determine whether to accept the change or ignore it.

At first glance, a simple `uniform` function would seem sufficient. However, since the `genes` and the `idup` contain an inequal number of accumulators, simply applying `uniform` would not give each accumulator an equal chance of being

selected. Instead, we consider how many accumulators are present in each. The genome contains n genes, each with u units (accumulators). The inverted duplication contains u bins, each with two units. Thus, if x is the number of genes, then the total number of units is proportional to $2 + x$ (the two being from the inverted duplication) while the number of units in the genes is proportional to x. Thus, the probability of selecting a unit from the genes is $\frac{x}{2+x}$.

The functions `chgGenes` and `chgIDup` apply one change to either a list of duplications or a list of bins (an inverted duplication), respectively. The location of the change is a uniform distribution over all possible sites.

```
mkModel :: Float -> Model
mkModel gp v = enumTT [1-p,p] [genes,idup]
               where genes = transAt idT (chgGenes gp) 2
                     idup  = transAt chgIDup idT 1
                     x     = fromIntegral v
                     p     = x/(2+x)
```

A model that accepts all changes is defined by `var` and a model that accepts only one-third of changes to the genes is defined by `fam`.

```
var :: Model
var = mkModel 1

fam :: Model
fam = mkModel (1/3)
```

The state of interaction is defined as a function on the genome. The possibilities for interaction are `Loss`, `None`, `Full` and `Partial`.

```
data Interaction = Loss | None | Partial | Full
```

The state of `Loss` occurs when the pairs of the inverted duplication lined up sequentially had no pattern where the sum of changes between one sense and anti-sense was less than 11, the sum in the next less than 6, and the sum in the next less than 11. In other words, we rolled a 10-5-10 upper bound across the inverted duplication, and if no match was found, it was considered lost.

```
match x y z = x <= 10 && y <= 5 && z <= 10
```

The function `defunct` determines if an inverted duplication has been lost. This function takes three sequential pairs from an inverted duplication. Each pair `(si,ai)` consists of a sense `si` and anti-sense `ai`, which are represented as units accumulating changes. If the sum of the changes in the first pair and the third pair are less than or equal to 10, and the sum of the changes in the second (middle) pair is less than or equal to 5, then the inverted duplication is not defunct (not lost), so the function returns `False`. If the first three pairs do not, however, match the 10-5-10 pattern, then function shifts one pair down the

sequence and looks again. If the function reaches the end of the sequence of pairs, and no sequence of three matching the pattern is found, the inverted duplication is considered lost. Implicitly, this means that all simulation models must have at least three units to be interesting.

```
defunct ((s1,a1):(s2,a2):(s3,a3):sx) |
        match (s1+a1) (s2+a2) (s3+a3) = False
defunct (_:sa2:sa3:sa) = defunct (sa2:sa3:sa)
defunct _ = True
```

If the inverted duplication is not lost, we proceed to inspect each gene to see if it interacts with the inverted duplication. Such interaction is determined by adding the changes in each unit in the gene to the anti-sense unit in the associated pair of the inverted duplication. If the sum is less than 5 for any unit, the gene is considered to interact with the inverted duplication.

```
interact :: IDup -> Dup -> Bool
interact i d = any (<=4) $ zipWith (+) (map snd i) (drop n d)
               where n = length d-length i
```

Gene interaction is tested for all genes, and the genome interaction state is determined by comparing the number of genes which interact with the total number of genes. If all genes interact, interaction is `Full`. If no genes interact, interaction is `None`. If some genes interact, interaction is `Partial`.

In this case, we define `interaction` as a function from a genome to an interaction state. The function `interaction` takes a `Genome`, which is a pair consisting of an inverted duplication `i` and a sequence of genes `gs`. The function `defunct` determines if the given inverted duplication is lost. If so, the interaction function always returns `Loss`. Otherwise, the number of genes `g` is determined by computing the length of the list `gs`, along with the number of genes currently interacting with the inverted duplication, which is determined by filtering the sequence of genes to retain only those that interact, and then counting them. These two values are then used to determine the interaction state as `None`, `Partial` or `Full` as described above.

```
interaction :: Genome -> Interaction
interaction (i,gs) | defunct i = Loss
                   | l==0      = None
                   | l==g      = Full
                   | otherwise = Partial
                       where l=length (filter (interact i) gs)
                             g=length gs
```

For each simulation run, we start with a genome that consists of an inverted duplication with no changes and a list of genes with no changes. We selected one of these genes to be the *founder gene* and set it aside. The remaining genes accumulated a given number of initial changes spread among them. The function

`g` creates a `Genome` given an initial chance of changes `c`, the number of units per gene `u` and the number of genes `n`. This function first constructs the inverted duplication and genes with 0 changes. A list of $n-1$ of genes is constructed, which has the requested changes randomly applied. The function `chgGenes` here is the same as above; it applies one change per call to the given list of duplications. The parameter `1` indicates that it should not discard any changes. The founder gene, with no changes, is appended. This completes the creation of the genome. Once the genome is created, the model transition can be applied iteratively to produce a trace of the evolution.

```
g :: Float -> Int -> Int -> R Genome
g c u n = do gs' <- (m *. (random $ chgGenes 1)) gs
             return (zip f f,f:gs')
             where m  = round (fromIntegral n*c)
                   f  = list u 0
                   gs = list (n-1) f
```

Note the use of `random` to ensure that the change will produce a single randomized value rather than a distribution. This change is then iterated to select many randomized values, thus producing a randomized distribution, approximating the actual distribution.

We found that running a full simulation of the genome used tremendous amounts of memory and time, so we opted for randomized simulations, allowing the biologists to trade off between detail and time. In order to minimize memory usage, we performed the aggregation of traces at the outermost level. This avoids constructing a distribution during each simulation run, holding instead only a single randomized genome which is built into a randomized trace.

Changes were applied using the model until the interaction entered the state of `Loss`. Since these were randomized changes, we only accumulated an `RTrace`, which we then put together over many runs to produce an `RSpace`. We then analyzed each distribution to count how long the simulation stayed in partial interaction, as this was the configuration the biologists found interesting.

```
sumDiff :: [Dist Interaction] -> Float
sumDiff ds = sum (map (prob2Float . ((==Partial) ??)) ds)
```

We can then simply divide by the number of runs in the space to find the average time spent in interaction, which we can plot for varying models and number of genes. An example of the results is shown in Figure 3.

MicroRNAs are significant in determining the function of genes. However, it is not completely clear how about microRNAs have evolved—in particular, biologists note that microRNAs are not present with equal likelihood in all genes. Our model makes two concrete predictions about the presence of microRNAs: First, microRNAs are more likely to be found in "variable" genomes rather than "family" genomes, and second, as a probability per gene, microRNAs are more likely to be found in organisms with smaller genomes. Preliminary experimental results discussed in the forth-coming paper [1] supports both of these predictions.

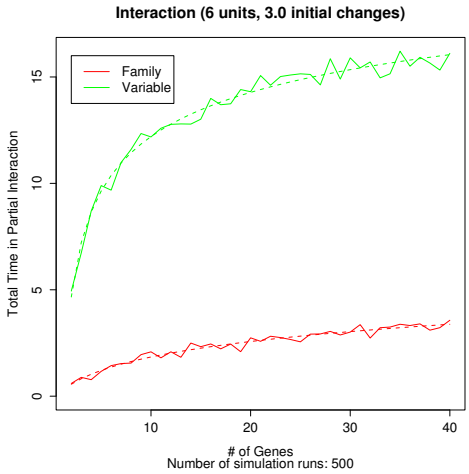

Fig. 3. Simulation results

6 Related Work

Work has been done on both probabilistic programming and modeling biological systems.

A theoretical, set-oriented treatment of probabilistic computations is given in [7]. The author points out that probabilistic computations can be considered in a monadic domain.

A monadic probability implementation is demonstrated by [14]. The authors show how probability distributions can be constructed using transitions similar to our own. The transitions can be combined monadically and operators are used to derive expected values and take samples. The authors also demonstrate a formal stochastic lambda calculus for representing probabilistic computations.

A randomized probabilistic language is demonstrated by Park, et al. [12]. Their method is based on sampling from probabilities, which can then be combined to form random results or probability distributions. Their method involves repeated sampling of the probability space, whereas our method can concretely represent this problem with deterministic probability distributions to find an exact probabilistic result.

An early attempt to model biological systems was done by McAdams and Shapiro [9]. The authors compared biological systems to electrical circuits noting that, like electrical circuits, biological systems operate in parallel and switches may describe activation or repression of either electricity or biological function.

Sato and Kameya [17] introduce a statistical logic learning language called PRISM based on Prolog. This language is designed for modeling uncertainty at a high level and can also infer parameters based on a set of given data.

A mathematical approach was taken by Nilsson and Fritzson with the Modelica system [11]. Modelica is an equation-oriented programming environment,

which includes objects, allowing a direct modeling of biological components and the continuous mathematical models that direct their behavior. The authors also allow the introduction of thresholds, which allow discrete events to be modeled based on continuous value equations. A graphical environment exists, which allows straightforward access by mathematically trained scientists to develop Modelica models.

Regev et al. [15] introduce an abstraction method for representing biological components as units of computation. They call these components *ambients*. An ambient is an isolated computation environment which may contain, in a hierarchical fashion, other ambients. The authors also describe complex, multi-level models which include functions at the molecular, cellular, and anatomical level. These situations are modeled by having a set of ambients for each level of detail, and using the hierarchy to specify the range of influence. A language, BioSpi, is briefly described which includes the concept of ambients and is designed for systems biology simulations.

Eker et al. introduce a method they called "pathway logic" [4], which is an algebraic approach that allows analysis of the abstractions. For example, the authors point out that the equality of $(x+y)*(x-y)$ and x^2-y^2 could be checked numerically for many possible values, but it can also be derived using a set of algebraic rewrite rules, which could form a proof. The authors define a specific set of rewrite rules involving proteins and cells and then show how analysis can provide several possible classes of results: explicit simulation, determining what constraints a given start state has on all future states (for example, if some property P is true, do we always reach a state that satisfies property Q?) and meta-analysis, which asks broadly which classes of starting states would satisfy some final criteria, thus allowing model disambiguation using actual data.

Pathway Modeling Language (PML) is introduced by Chang and Sridharan [3]. This language is based on the concept of binding sites—where two components have a compatible connector and so bind, allowing some private interactions and transformations, and then break apart with new connectors ready to bind to other components. They also provide for compartmentalization of reactions. This approach allows an event-oriented design where reactions happen as all preconditions are met and binding occurs. In this way, the order of reactions does not need to be explicitly specified.

7 Conclusions

High-level declarative languages, extended by suitable domain-specific abstractions, offer a great potential as executable modeling languages for scientists, because they support the incremental development of scientific models that can be instantly tested and easily revised and adapted.

We believe that typed functional languages are particularly well suited for this task since they allow the creation of type structures that closely reflect the modeled domains. This aspect gains in importance as scientific models evolve from being low-level and based on plain numbers toward incorporating higher-

level (data) structures, such as sequences, tuples, and other data types, as evidenced by the presented application from genome evolution.

References

1. E. Allen, J. Carrington, M. Erwig, K. Kasschau, and S. Kollmansberger. Computational Modeling of microRNA Formation and Target Differentiation in Plants. 2005. In preparation.
2. J. C. Carrington and V. Ambros. Role of microRNAs in Plant and Animal Development. *Science*, 301:336–338, 2003.
3. Bor-Yuh Evan Chang and Manu Sridharan. PML: Toward a High-Level Formal Language for Biological Systems. In *Bio-CONCUR*, 2003.
4. Steven Eker, Merrill Knapp, Keith Laderoute, Patrick Lincoln, Jose Meseguer, and Kemal Sonmez. Pathway Logic: Symbolic Analysis of Biological Signaling. In *Pacific Symp. on Biocomputing*, pages 400–412, 2002.
5. M. Erwig and S. Kollmansberger. Probabilistic Functional Programming in Haskell. *Journal of Functional Programming*, 2005. To appear.
6. Giry, Michèle. A Categorical Approach to Probability Theory. In Banaschewski, Bernhard, editor, *Categorical Aspects of Topology and Analysis*, pages 68–85, 1981. Lecture Notes in Mathematics 915.
7. Jones, Claire and Plotkin, Gordon D. A Probabilistic Powerdomain of Evaluations. In *4th IEEE Symp. on Logic in Computer Science*, pages 186–195, 1989.
8. A. J. Lotka. The Growth of Mixed Populations: Two Species Competing for a Common Food Supply. *Journal of Washington Academy of Sciences*, 22:461–469, 1932.
9. Harley H. McAdams and Lucy Shapiro. Circuit Simulation of Genetic Networks. *Science*, 269(5224):650–656, 1995.
10. Morgan, Carroll and McIver, Annabelle and Seidel, Karen. Probabilistic Predicate Transformers. *ACM Trans. on Programming Languages and Systems*, 18(3):325–353, 1996.
11. Emma Larsdotter Nilsson and Peter Fritzson. Using Modelica for Modeling of Discrete, Continuous and Hybrid Biological and Biochemical Systems. In *The 3rd Conf. on Modeling and Simulation in Biology, Medicine and Biomedical Engineering*, 2003.
12. Park, Sungwoo and Pfenning, Frank and Thrun, Sebastian. A Probabilistic Language based upon Sampling Functions. In *32nd Symp. on Principles of Programming Languages*, pages 171–182, 2005.
13. PFP. Probabilistic Functional Programming Library, 2005. `http://eecs.oregonstate.edu/~erwig/pfp`.
14. Ramsey, Norman and Pfeffer, Avi. Stochastic Lambda Calculus and Monads of Probability Distributions. In *29th Symp. on Principles of Programming Languages*, pages 154–165, 2002.
15. Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Shapiro. BioAmbients: An abstraction for biological compartments. *Theoretical Computer Science, Special Issue on Computational Methods in Systems Biology*, 325(1):141–167, September 2004.
16. Renshaw, Eric. *Modelling Biological Populations in Space and Time*. Cambridge University Press, 1993.
17. T. Sato and Y. Kameya. Parameter Learning of Logic Programs for Symbolic-Statistical Modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.