Modeling Biological Systems with FuSE

Martin Erwig and Steve Kollmansberger
School of EECS
Oregon State University
[erwig,kollmast]@eecs.oregonstate.edu

November 17, 2005

1 Introduction

This manual introduces a tool to support the computational modeling of biological systems on a high level of abstraction. The basic idea of the chosen approach is to represent a biological system as a set of objects and a set of functions that operate on these objects. Any system defined in this way can be simulated and the results can be visualized.

The presented tool is implemented as an extension of the functional programming language Haskell in form of a library, which specifically provides support for the definition of probabilistic functions and simulations. The library extension is called FuSE, which stands for Functional Simulation Environment.

The defined functions provide a black box operation without requiring detailed programming knowledge or prior knowledge of Haskell. However, since definitions of objects and functions will be expressed in Haskell, a basic understanding of Haskell is required to be able to effectively develop models

Rather than ponder the details of our implementation, we instead provide concrete examples of functions which use our libraries and how to construct them. These functions provide probabilistic simulations and results, which our library can convert into R programs providing visualizations.

To be able to successfully use the described modeling approach, a basic understanding of the following concepts is required.

- Values, functions, and types in Haskell
- Probability distributions and transitions
- Basic operation of R

In this manual, we will introduce all three of these concepts, along with a very simple biological application. In the next section, we will demonstrate some basic elements of Haskell. Section 3 provides some background information on the notion of types. In Section 4 we will introduce the concept of probabilistic

functions and show how to simulate probabilistic events over time. In Section 5 we will show how the results of a simulation can be graphed in various ways using the R statistic package. In Sections 6, we show how to employ randomizations in simulations. Section 7 discusses in more detail how to define models that describe time- or generation-dependent phenomena. In Section 8 we will illustrate how to employ more sophisticated Haskell data types to represent more structured models and how to employ probabilistic data types in that context. These three sections will provide the necessary background to discuss the miRNA evolution model in Section 9.

2 Defining Values and Functions in Haskell

The simulation library makes designing and implementing simulations in Haskell fairly straightforward. In this document, we will discuss the features necessary to construct useful simulations from the ground up, using examples. Source code should be entered in a text editor and placed in files with the extension `.hs`. Programs should be run in GHC by loading GHCi with the source file. For example, the command

```
ghci Tree1.hs
```

start the GHCi Haskell interpreter and loads the first example program `Tree1.hs`, which is shown in Figure 1.

Once GHCi is loaded, it will report any errors found in the source file. These must be corrected before any evaluation can be attempted. Once the file loads without errors, functions may be evaluated by entering their name and any parameters at the prompt.

We will start by considering a simple example. Consider the case of a tree which starts at five feet tall and grows by one foot every year. To define this, we first need to find the key components. Units are not relevant in Haskell except in relation to each other. For example, if you are considering kilometers and meters, it is important to know that one is 1000 of the other, but not precisely what the units are. In this case, we have a single tree with a height. No other information about the tree is given, so there is no need to consider, for example, whether the tree is deciduous or evergreen.

Since we want to define a model that yields data over time, we we have to define the initial condition or value and how the value changes over time. The initial case is the height of 5. This initial value can be represented with a value definition, which consists of a name for the value and the value itself.

```
height = 5
```

The name `height` is simply an arbitrarily chosen label that has no extrinsic meaning to the program. In particular, it does not affect how the program operates nor does it cause the program to catch potentially meaningless operations.

```
module Tree1 where

height = 5
grow h = h + 1
```

Figure 1: The Haskell program `Tree1.hs`.

The value change is defined by the rule that each year the height of the tree is one more than the previous year. This change can be represented by a function. Function definitions look very similar to value definitions. The only difference is that functions take one or more parameters, which are given in the definition by names that follow the function name. These parameters can then be combined in various ways to produce the result. In this case, we want to define a grow function that takes one parameter, the height, and returns the height increased by one.

```
grow h = h + 1
```

In this case, the function is named `grow` and it takes one parameter, `h`, which symbolizes the height. It returns the value of `h + 1`. The definition is the same as a mathematical function such as $f(x) = x + 1$.[1]

We save the complete program into a file called `Tree1.hs`. This file contains the two definitions already shown and a `module` declaration line that gives the program a name, see Figure 1.

We then load this program in GHCi by issuing the shell command `ghci Tree1` or `ghci Tree1.hs`.

```
$ ghci Tree1.hs

...

Loading package base ... linking ... done.
Compiling Main             ( Tree1.hs, interpreted )
Ok, modules loaded: Main.
*Tree1>
```

At this point we are ready to begin evaluating functions. We can query the height of the tree by entering `height`, which accesses that constant.

```
*Tree1> height
5
```

If we want to know what happens in the subsequent year, we can apply the function `grow`.[2]

[1] Note that the parentheses around the argument x are not really needed and are therefore usually omitted from function definitions in Haskell programs.

[2] Again, note that no parentheses are needed around the argument since there is no ambiguity as to what the argument to the function `grow` is, unlike in the next example.

```
*Tree1> grow height
6
```

We can apply `grow` an arbitrary number of times to simulate multiple years in the future.[3]

```
*Tree1> grow (grow height)
7
```

To leave GHCi enter `<Control-D>` or type:

```
:q
```

followed by `<Return>`, which will bring you back to the command line.

3 Types

Haskell is a strongly typed language, which means that all values and functions of a program must have a precisely determined *type*. The simulation environment that is built on top of Haskell inherits this property. Therefore, a basic understanding of what types are is very helpful to know for every user.

3.1 Type Signatures

Similar objects or values are grouped into collections, called *types*. Any member, say `x`, of such a collection, say `T`, is said to *have* or *be of* the type `T`, which is written as:

```
x :: T
```

For example, integer values have the type `Int`, and thus we write, for example, `5 :: Int`. When we assign a name to a value in Haskell, as we did with the definition for `height`, that name has the same type as the value it represents. Therefore, we have:

```
height :: Int
```

Such statements about the type of a value or name is also sometimes called a *type signature*. These type signatures can be placed in Haskell programs. Although in most cases, Haskell does not require type signatures to be given explicitly (since the Haskell compiler can figure out the types of all objects automatically), it is nevertheless a Good Thing to provide type signatures for

[3] Here we have to enclose the argument of the leftmost `grow` function by parentheses because function application is a left-associative operation, that is, $f\ g\ x$ is *not* parsed as $f(g(x))$, but rather as $(f(g))(x)$. Therefore, we need parentheses around the innermost application since we want to apply `grow` to the result of applying `grow` to `height`. In contrast, the expression `grow grow height` would try to apply `grow` to `grow`, which doesn't make sense and would be prevented by Haskell by reporting a type error; see also Section 3.

all definitions in a program, because it documents the program and supports debugging and maintenance.

A type expresses that certain operations are allowed on all its members while others are forbidden. For example, integer values can be added, but they cannot be applied as functions on other objects.

3.2 Function Types

Functions are also values and thus have a type. For example, the function `grow` has the type `Int -> Int`, which expresses the fact that `grow` accepts one parameter of type `Int` and returns a value that is also of type `Int`. The type signature for `grow` thus reads as follows.

```
grow :: Int -> Int
```

The type information about individual objects is used by the *type checker* of Haskell to ensure that objects are combined only in a correct way. (The set of rules that define how object can be composed are called the *type system*.) For example, one important rule says that a function of some type `T -> T'` can be applied only to objects of type `T`, which makes much sense because the function type expresses that the function only accepts objects of type `T`. The rule for function application also says that if a function of type `T -> T'` is applied to an object of type `T`, then the result will be a value of type `T'`.

For example, it is type correct to apply `grow` to `height` since `height :: Int`, which matches the argument type of `grow`. Moreover, the result of this application is also of type `Int`, that is, `grow height :: Int`. The typing rule for function application also explains why, for example, the expression `grow grow height` (without the parentheses around `grow height`) is *not* correct. Since function application is left associative, that expression is interpreted as `grow grow` applied to `height`. Now `grow grow` is not type correct because the argument type of `grow` must be of type `Int`, but the argument provided is of type `Int -> Int`.

What are the types of functions hat have more than one parameter? Consider, for example, the predefined minimum function `min` that takes two numbers and returns the smaller of the two. The type of `min` is written as follows.

```
min :: Int -> Int -> Int
```

¿From the typing rule for function application it follows that if `min` is applied to two `Int` values, it will return another `Int` value. A function can be applied to multiple arguments by simply juxtaposing the function name to all arguments. In the case of `min` we can write, for instance:

```
*Tree1> min height 7
5
```

3.3 Type Definitions

In addition to using the built-in types, it is also a good practice to introduce type definitions to document an application. For example, the fact that we are using `Int` values to represent tree heights can be captured by defining a corresponding type.

```
type Height = Int
```

Now instead of the previous signature for the value `height` we can use the following one.

```
height :: Height
```

Since the type `Int` is typically used to represent many different, unrelated values, defining and employing more specific types, such as `Height`, helps to document simulations and also aids with debugging.

We can also use the defined type to give a more meaningful type signature for the function `grow`.

```
grow :: Height -> Height
```

Note that the shown form of type definition introduces just a new name for an existing type, that is, `grow` can still be applied to arbitrary `Int` values. In fact, the type definition says that any `Height` value *is* an `Int` value.

4 Building a Simple Simulation

Continuing our tree growth example, it may not always be the case that the tree grows one foot per year. For example, some years the tree may not grow appreciably. We can define a straightforward function for the case when a tree does not grow in a given year.

```
notGrow :: Height -> Height
notGrow h = h
```

We then have two growth functions, `grow` and `notGrow`, which we can combine to create a probabilistic growth function. Assume there is an 80% chance that a tree grows by one foot a year; otherwise, its height stays unchanged. We can represent this probabilistic function using an enumeration constructor, which allows one from a group of functions to be selected with given probabilities.

```
probGrow :: Height -> Dist Height
probGrow = enumT [0.8, 0.2] [grow, notGrow]
```

In this case, we have constructed a probabilistic function which has an 80% chance of being the function `grow` and a 20% chance of being the function `notGrow`. The defined function `probGrow` itself maps an integer into a collection

```
module Tree2 where

import Probability

type Height = Int

height :: Height
height = 5

grow :: Height -> Height
grow h = h + 1

notGrow :: Height -> Height
notGrow h = h

probGrow :: Height -> Dist Height
probGrow = enumT [0.8, 0.2] [grow, notGrow]
```

Figure 2: The revised tree model `Tree2.hs`.

of integer values each with an associated probability, representing the different possible tree heights and their likelihood. The type of such a collections of `Int` values tagged with probabilities is `Dist Int`, which is equivalent in the current example to `Dist Height`.[4]

In order to use probabilistic functions, we need to include the probability library. Here is the new code, which is contained in the file `Tree2.hs`, which is shown in Figure 2.

We can load this program into GHCi as before, but now we can employ the probabilistic function `probGrow`. After one year, the tree will have grown by one foot with an 80% probability. We find this result by applying `probGrow` to `height`.

```
*Tree2> probGrow height
6  80%
5  20%
```

If we wish to apply the probabilistic growth function multiple times, we cannot simply apply `probGrow` to the result of another `probGrow` application as we did before with the function `grow`. Why? Because the types do not match: Remember that `probGrow` requires an integer as an argument (recall that `Height` is just another name for the type `Int`), whereas the application of `probGrow` yields a distribution of integers. Moreover, it is not really clear what it should mean to apply an integer function to a distribution of integers. What we really

[4] A probabilistic function is also called a *transition*. In general, a transition takes some value of type `T` and produces a distribution of possible new values of the same type. In general, whenever we see a type `T -> Dist T` for any type `T`, we call this a transition on `T`, and represent it with the type `Trans T`. For example, the function `probGrow` has type `Height -> Dist Height`, which we can alternatively consider as `Trans Height`, a transition on heights.

want to have is that the transition `probGrow` is applied to each value in the distribution combining the probabilities of the input with the probabilities of the values from the computed result distribution. The library provides such a function called `*.` which will repeatedly apply a probabilistic function a given number of times on some input. We can find out the height of the tree after, for example, five years, as follows.

```
*Tree2> (5 *. probGrow) height
 9  41%
10  33%
 8  20%
 7   5%
 6   1%
 5   0%
```

Notice that the results are presented sorted by likelihood, with the most likely results shown first. Also note that the probabilities are rounded to 0 decimal places by default.

5 Visualization

In many cases, it can be easier to understand and compare distributions when they are graphed, as opposed to simply a list of probabilities. Our library provides graphing functionality through the statistical package R. Once you have the distribution, or distributions, that you want to graph, the library provides functions that can produce R files to visualize the distributions. We provide access to a number of R graphical customizations to make the resultant graphs both useful and aesthetic.

5.1 Figures and Plots

Continuing the tree growth example, assume we want to plot the distribution of tree heights after five years. We have previously seen that the command `(5 *. probGrow) height` produces this distribution. We can graph it in a straightforward way using the `plotD` and `fig` functions. First, we have to load the program `Tree3.hs` into GHCi, which ensures that the module defining the visualization functions is loaded. Then we can try to create a figure containing one plot as follows. The `plotD` function takes a distribution and creates a plot out of it. A plot is a single line with attributes such as color and line type. A list of one or more plots can be composed into a figure by the `fig` function. In the shown example, the list contains just one plot.

```
*Tree3> fig [plotD ((5 *. probGrow) height)]
*Tree3>
```

Although no result occurs in the interactive GHCi session, a file `FuSE.R` has been created that can be opened in R. Once R is started, we can load the file

from R's interpreter as follows.

```
> source("FuSE.R")
```

At this point, a graph, like the one shown in Figure 3, should appear on the screen.

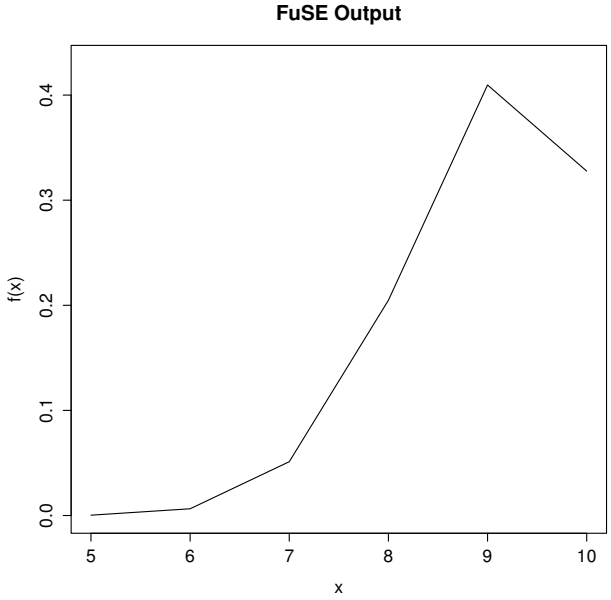

Figure 3: Five Year Plot

At this point, the graphic looks rather plain and boring. Later, we will show how to customize titles, labels, and colors and how to define a legend.

Although this graph effectively shows the tree height at five years, how does this compare to the height at ten or fifteen years?

We can use similar commands, namely `(10 *. probGrow) height` and `(15 *. probGrow) height` to find out the distribution at ten and fifteen years. In order to simplify our task slightly, we can define a new function which produces a distribution of heights at a given number of years.

```
type Year = Int

heightAtTime :: Year -> Dist Height
heightAtTime n = (n *. probGrow) height
```

To clearly document what the function does, we have introduced another type definition for years. The function `heightAtTime` simply substitutes whatever

number it is given into the function call we have already determined for computing distributions. Since our goal is to eventually turn distributions into plots placed in a figure, we can go one step further and define `heightAtTime` to directly produce a `Plot` value instead of a distribution.

```
heightAtTime :: Year -> Plot
heightAtTime n = plotD ( (n *. probGrow) height)
```

Now we can use the function `fig` to produce a figure with, say, three, plots.

```
fig [heightAtTime 5, heightAtTime 10,heightAtTime 15]
```

We can simplify the above expression even further by using the Haskell function `map` that allows a function to be repeatedly applied to a list of values. In the present example `heightAtTime` is applied to three values. Instead of writing the application three times, we can use the following shorter form

```
*Tree3> fig (map heightAtTime [5,10,15])
*Tree3>
```

We see that `map` is a function that takes two arguments, a function (here: `heightAtTime`) and a list of values whose type match the argument type of the function argument (here: `[5,10,15]`), and applies the function to each element of the list producing a list of the results of each application (here: the list of plots for 5, 10, and 15 years). This list is then used as an argument for the `fig` function.

The figure is again created as an R file called `FuSE.R`, which can be loaded into R to produce the graph shown in Figure 4.

This graph, however, could be confusing. Which lines refer to which years? The data is fundamentally being represented, but it is not annotated in a clear way.

5.2 Customizing Visualizations

We allow the user to set options on how they would like the plots and figure rendered. In this case, we would like to draw each curve in a different color, and give it a corresponding label in the legend. We use the following curly bracket notation to assign options to individual plots. For example, to give the plot of five years the color of blue and the label "5 Years", we modify it to be:

```
(heightAtTime 5){color=Blue,label="5 Years"}
```

Note the added parenthesis around `heightAtTime 5`. If these were not present, the options would attempt to apply to the integer value 5, which does not make sense since an integer is not a plot with customizable options. If we likewise assign green to ten years and red to fifteen years, we can create a figure with three plots as follows.

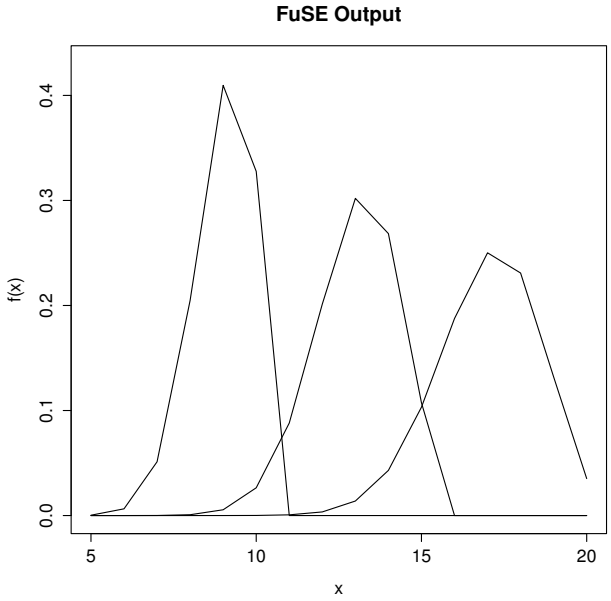

Figure 4: Five, Ten, and Fifteen Year Plot

```
fig [(heightAtTime 5){color=Blue,label="5 Years"},
     (heightAtTime 10){color=Green,label="10 Years"},
     (heightAtTime 15){color=Red,label="15 Years"}]
```

Again, this requires a lot of typing, and again, we can simplify this expression by introducing another function that creates a plot given a year and a color. To put the year (which is a number) into the label (which is a string), we employ the `show` function that can convert numbers (and other values) into strings. We then connect the two strings with the concatenation function `++`. The function uses the tuple notation to accept a pair of values, in this case a year and a color. Alternatively, we could have defined `heightCurve` as a function of two parameters (i.e. with type `Year -> Color -> Plot`). However, the given definition facilitates the mapping across a list of arguments.

```
heightCurve :: (Year,Color) -> Plot
heightCurve (n,c) = (heightAtTime n){color=c,label=show n++" Years"}
```

To create the figure, we wish to create a list of plots by applying our `heightCurve` function to a series of year and color tuples. As we have already done with `heightAtTime`, we can employ the `map` function to create a list based on applying a function to each element in a list. In this case, we apply the `heightCurve` function to a list of color and year tuples, which creates a list of plots, precisely what is needed by the `fig` function.

```
map heightCurve [(5,Blue),(10,Green),(15,Red)]
```

In addition to customizing the appearance of the plot lines, we can use, instead of `fig`, the function `figP`, which allows a set of figure-wide parameters to be specified. These parameters include a general title and labels for the x and y axes. It is also possible to customize the output filename with these options.

In this case, we want to set the general title to "Tree Growth", and label the x axis with "Height" and the y axis with "Probability". We achieve these changes by modifying the default figure environment, called `figure`, with the same curly bracket notation that we have already used for modifying individual plot settings. We resultant expression is shown here.

```
figP figure{title="Tree Growth",
            xLabel="Height (ft)",
            yLabel="Probability"}
     (map heightCurve [(5,Blue),(10,Green),(15,Red)])
```

To save the typing this long expression into the GHCi interpreter, we have placed a definition `fig3 = ...` into the program `Tree3.hs`, which enables the creation of the figure by just entering that name. We also define the value `growthFig` as the shown customized figure parameter setting to reuse it for other plots to be produced later.

```
*Tree3> fig3
*Tree3>
```

Again, the result is an R file that can be loaded from the R interpreter. Plotting the results of this function gives us a very presentable result shown in Figure 5.

For cases in which the actual choice of the colors doesn't matter, we also provide the function `autoColor` that allows a very quick creation of multiple distinguishable plots in one figure. For example, the expression

```
fig (autoColor (map heightAtTime [5,10,15]))
```

creates an output that looks, except for the customized labels, exactly like Figure 5. The complete program `Tree3.hs` is shown for reference in Figure 6.

6 Employing Randomization in Simulations

The examples shown thus far elucidate the core of our simulation library. Although the examples are purposefully simple, they can be scaled to produce more complex results with more details. In this and the following two sections we will explore different directions to extend simple models.

In this section we address the problem that the number of elements in a distribution grows exponentially at each step. In a short number of steps, the distribution will be too large to fit in memory, or too slow to manipulate. In these cases, we can use randomization to select elements from distributions and

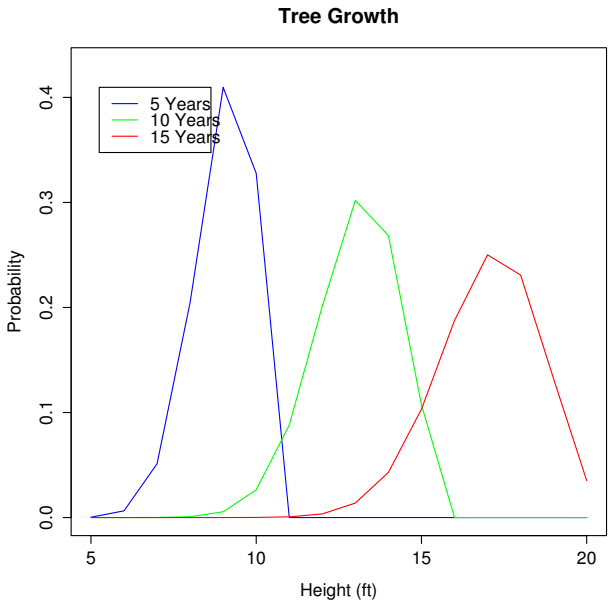

Figure 5: Five, Ten, and Fifteen Year Plot, with Annotations

use such representative samples to approximate the results in tractable time and space.

We start by introducing the concept of a randomized values in Section 6.1. We then introduce how to produce randomized distributions and transitions, and their application to simulation in Section 6.2.

6.1 Randomized Values

Remember that a distribution (of type, say `Dist T`) is basically a collection of `T` values with attached probabilities that describe all the different possible results of a value or function and how likely they are to occur. The `pick` function allows the selection of exactly one value from such a collection, randomly yielding one of the values according to the specified probability. The result of applying `pick` to a distribution is not simply a `T` value, but a *randomized* `T` value. In particular, different applications of `pick` to the same distribution might yield different results. This fact is reflected in the type of the returned value, which is `R T` instead of `T`.

Randomized values require a special treatment to be printed. If we just apply `pick` to a distribution, the randomly selected value will *not* be printed.

```
*Tree3> pick (probGrow height)
*Tree3>
```

```
module Tree3 where

import Probability
import Visualize

type Height = Int
type Year = Int

height :: Height
height = 5

grow :: Height -> Height
grow h = h + 1

notGrow :: Height -> Height
notGrow h = h

probGrow :: Height -> Dist Height
probGrow = enumT [0.8, 0.2] [grow, notGrow]

heightAtTime :: Year -> Plot
heightAtTime n = plotD ((n *. probGrow) height)

heightPlot :: (Year,Color) -> Plot
heightPlot (n,c) = (heightAtTime n){color=c,label=show n++" Years"}

growthFig = figure{title="Tree Growth",
                   xLabel="Height (ft)",
                   yLabel="Probability"}

fig3 = fig [heightAtTime 5]

fig4 = fig (map heightAtTime [5,10,15])

fig5 = figP growthFig (map heightPlot [(5,Blue),(10,Green),(15,Red)])

fig5b = fig (autoColor (map heightAtTime [5,10,15]))
```

Figure 6: Using visualization with the tree model `Tree3.hs`.

However, we can expliticly demand the printing of randomly selected values using a `printR` function as follows.[5]

```
*Tree3> printR $ pick (probGrow height)
6
```

Now we can observe that the repeated evaluation of the same expression yields

[5]The reason for the additional syntax has to do with Haskell's type system and how random numbers are generated through a monad. An understanding of the background is not needed to use randomized values.

different results. [6]

```
*Tree3> printR $ pick (probGrow height)
5
*Tree3> printR $ pick (probGrow height)
6
*Tree3> printR $ pick (probGrow height)
6
```

About 80% of the time, we should see a 6, and a 5 the rest of the time. We can use the function `random` to turn a transition into a randomized function which generates a single value from the resultant distribution. Such a function is called a `RChange` because it takes one value and produces one randomized value.

6.2 Randomized Transitions

Recall that a normal transition is a function of type `a -> Dist a` that produces a distribution of elements from one input. We also use the type abbreviation `Trans a` for a normal transition. As already mentioned, the type `R a` indicates when a value is random and may be different from run to run. A randomized transition produces a distribution of values that may change from run to run—representing an approximation of a transition. This is similar to the type `Trans`, so we define the type `RTrans`.

```
type RTrans a = a -> RDist a
```

Usually, such a transition is constructed from a regular transition by repeatadly selecting one of the elements from the resulting distribution. This selection is performed with respect to the probabilities in the distribution.

In order to construct a randomized distribution, the system picks a specified number of items from a regular transition. This number of items can be considered a quality-computation tradeoff, with more computation converging the randomized distribution to the actual distribution.

We introduce the simulation function `~.` to produce a randomized transition from a regular transition.

```
(~.) :: Int -> Trans a -> RTrans a
```

The integer value indicates how many samples to take from the transition when producing the randomized distribution. We can assume, for example, that we want to take ten samples. We can create such a randomized transition as follows.

```
rProbGrow :: RTrans Height
rProbGrow = 10 ~. probGrow
```

[6] In particular, trying out these examples will generally lead to different sequences of 6s and 5s.

We'd also like to be able to find out what the height is at a particular given year. In order to accomplish this, we introduce the function `~*.`. This function combines the simulation and randomization aspect of `~.` and the iteration aspect of `*.`. Precisely, the function uses random selection to pick a single element for each step of the iteration, and accumulates the requested number of samples at the final step.

```
(~*.) :: (Int, Int) -> Trans a -> RTrans a
```

The two integer parameters represent the number of samples and the number of iterations, respectively. We can extend the `rProbGrow` function to allow us to take a sample at a given year in the function `rHeightAtTime`.

```
rHeightAtTime :: Year -> RDist Height
rHeightAtTime n = ((10,n) ~*. probGrow) height
```

Since these transitions consider only a fixed number of elements, they are much faster than the exponentially explosive deterministic computations. For example, attempting to deterministically compute the distribution of heights at year one hundred would previously have taken some time (at least several seconds). Using the randomized transition, however, is much faster.

```
*Tree4> printR $ rHeightAtTime 100
85  20.0%
86  20.0%
87  20.0%
78  10.0%
81  10.0%
83  10.0%
84  10.0%
```

A randomized distribution can be converted into a plot using the `plotRD` function, which behaves identically to the `plotD` function, except that it takes a randomized distribution. With these two functions, we can create a function to generate a randomized height curve.

```
rHeightCurve :: (Int,Year,Color) -> Plot
rHeightCurve (runs,n,c) = (plotRD (( (runs,n) ~*. probGrow) height))
        {color=c,label=show n++" Years" ("++show runs++" Runs)"}
```

Finally, we can create a graph that compares the deterministic curve with two random ones.

```
fig7 = figP growthFig
                [heightCurve (20, Blue),
                rHeightCurve (500, 20, Green),
                rHeightCurve (25, 20, Red)]
```

We will compare the results of a deterministic calculation of the height at year 20 with two simulations: one with five hundred simulation runs, and another with only ten. This shows how increasing the number of simulation runs increases the accuracy of the approximation. The result is shown in Figure 7. We see that as the number of runs increases, the general accuracy of the curve increases.

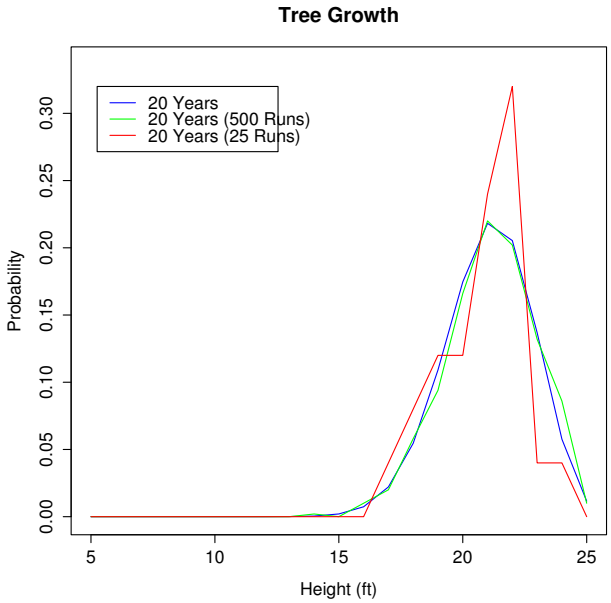

Figure 7: Twenty Years, Randomized vs. Deterministic

The advantage of the randomizing technique comes in the ability to control how much processing time is spent on the task. If greater accuracy is desired, the number of runs can be increased, and more time spent. If a quick result is desired, the number of runs can be decreased and the margin of error becomes wider.

7 Remembering Intermediate States Through Tracing

As simulation complexity increases, some computational aspects become difficult. If, for example, we wished to evaluate the growth of the tree at each year for one hundred years, it would be quite redundant to calculate it first for one year, then separately for two years, and again for three, and so on. Instead, we could calculate the growth for one hundred years and simply keep track of all intermediate results.

Tracing functions generate a list of all intermediate results obtained from a simulation. In deterministic simulations, this includes each distribution along the way. In randomized simulations, this is intermediate random value produced.

To facilitate tracing, we introduce the types `Trace` and `Space`, and their corresponding function types `Walk` and `Expand`. A `Trace` is simply a list of values, representing one value for each step. Since our transitions work in probability distributions, however, we actually need a distribution at each step. This is handled by a `Space`, which holds each intermediate distribution.

```
type Trace a = [a]
type Space a = Trace (Dist a)
type Walk a = a -> Trace a
type Expand a = a -> Space a
```

Each of these is also provided in a randomized form.

```
type RTrace a = R (Trace a)
type RSpace a = R (Space a)
type RWalk a = a -> RTrace a
type RExpand a = a -> RSpace a
```

Transitions can be converted into trace function using `*..` and `~...` These work just like `*.` and `~*.`, except that they record each intermediate step.

```
(*..) :: Int -> Trans a -> Expand a
(~..) :: (Int, Int) -> RTrans a -> RExpand a
```

For example, we can use `~..` to observe the growth of a tree over 10 years. Note that the most recent value is shown first. If we want to obtain only one sample per year, we set the sample rate to 1. In this example, the tree grew from the initial height of 5 steadily until 9, where by chance it stayed for four years, then grew further. Note that the syntax is identical to that of `~*.`. Since this produces a randomized value, we still need the `printR $` in order to display the results.

```
*Tree4> printR $ ((1,10) ~.. (random probGrow)) height
[12 100.0%
,11 100.0%
,10 100.0%
,9 100.0%
,9 100.0%
,9 100.0%
,9 100.0%
,8 100.0%
,7 100.0%
,6 100.0%
]
```

Traces are useful for plotting the development of values over time. The function `expected` takes a numeric distribution and computes the weighted average of it. We can take the expected value at each year of tree growth to plot what the expected height of the tree will be over time.

We can accomplish this by mapping the function `expected` to each element in the trace of the tree growth over time. This will produce a single curve which we can plot using the list plotting function `plotL`. We use `plotL` instead of `plotD` because we are not plotting probabilities here but values. Notice that x- and y-axis labels have changed to represent the new aggregate data being plotted. Since the data comes most recent first, but we want to plot it oldest first, we apply the `reverse` function to reverse the list.

```
figP figure{title="Tree Growth",xLabel="Year",
                yLabel="Height (ft)"}
                [plotL (reverse (map expected
                ( (15 *.. probGrow) height)))]
```

This alone would not be very interesting, so we combine it with a randomized run to compare the accuracy. We have purposefully set the number of simulation runs low so that differences between the actual expected value and randomized expected value can result.

```
aggregateTrace t = reverse (map expected t)

fig8 = figP figure{title="Tree Growth",xLabel="Year",
                yLabel="Height (ft)"}
                (autoColor
                [(plotL (aggregateTrace
                ( (15 *.. probGrow) height)))
                {label="Expected Growth"},
                (plotRL (mapR aggregateTrace
                        ( ( (5,15) ~.. probGrow ) height)))
                {label="Randomized Expected Growth"}])
```

Note that the function `mapR` is needed to generate the expected result for each distribution in the randomized trace. This is because the `aggregateTrace` function works on distributions, not randomized distributions. The function `plotRL` creates a plot given a randomized list. The result is shown in Figure 8. Using functions which aggregate distributions (such as `expected`) we can move from simply plotting probabilities to mining deeper results from our simulations.

8 Structured Modeling with Probabilistic Data Types

As simulations become reasonably complex, there will often be sets of highly organized data which need to be manipulated. In our example so far, we have

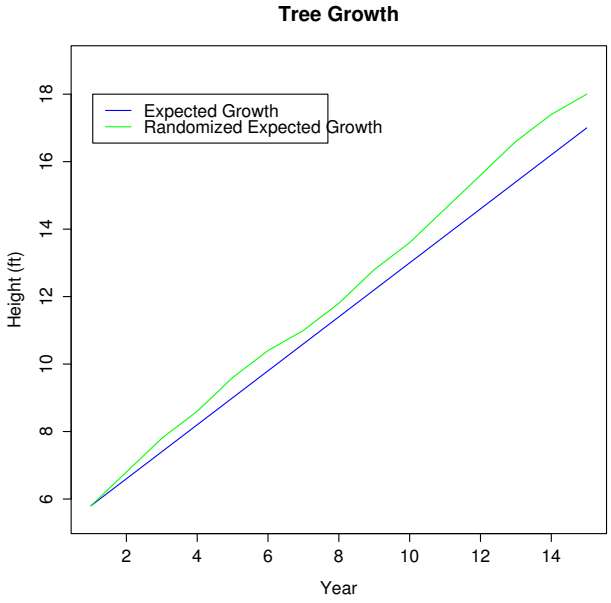

Figure 8: Expected Height Over 15 Years

manipulated a single number, the height of the tree. However, real simulations often have many structured collections of variables. Such simulations can be effectively represented through data structures, which can be acted on probabilistically in the same way as a simple number.

...

9 Case Study: A Model for the Evolution of micro RNAs

Biologists have determined that over generational time genomes experience evolutionary development. Part of this development includes genes from 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, as shown in Figure 9. Note that inverted duplication and duplication both represent DNS sequences. 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.

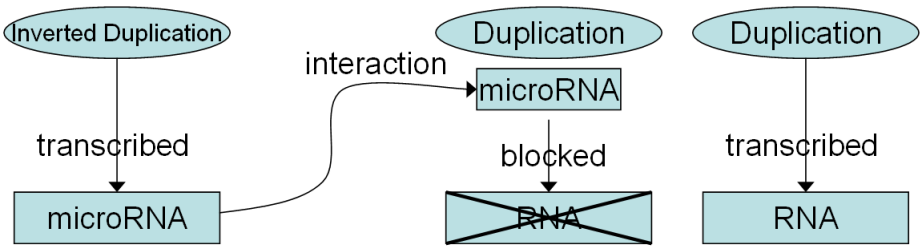

Figure 9: Effects of microRNAs on Gene Duplications

The genome consists of multiple genes, each of which is 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.

Our simulation used a genome consisting of `Dup`s and one inverted duplication `IDup`. An inverted duplication had three parts that could accumulate changes. These three parts arise from the fact that an inverted duplication is a strand of RNA folded onto itself. This can be viewed as two strands (called sense and anti-sense) and a loop. Changes in the loop are not significant for this model, and so it was not represented.

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])
```

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

acting, 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`.

```
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 integers giving the number of accumulated changes. If the sum of the changes in the first pair and the third pair is 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, the 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 pair, the gene is considered to interact with the inverted duplication. This concept is shown in Figure 10.

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

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,

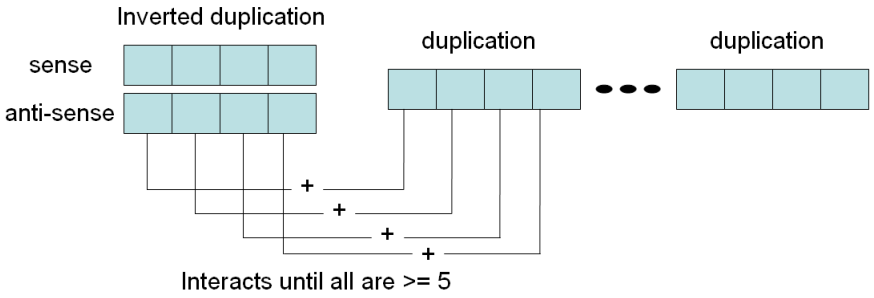

Figure 10: The Test of Interaction

interaction is `None`. If some genes interact, interaction is `Partial`.

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
```

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. 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 (`gp` $= \frac{1}{3}$) or variable (`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. In other words, the total number of units is $2u + u * n$. Of those, $2u$ units are in the inverted duplication and $u * n$ units are in the genes. Therefore, the probability of selecting a unit from the genes should be $\frac{u*n}{2u+u*n}$ which is the same as $\frac{n}{2+n}$.

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)
                     idup  = transAt chgIDup idT
                     n     = fromIntegral v
                     p     = n/(2+n)
```

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)
```

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 select one of these genes to be the *founder gene* and set it aside. The remaining genes

accumulate 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.

```
type NumUnits = Int
type NumIter = Int

g :: Float -> NumUnits -> NumIter -> 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. We use the randomization methods discussed earlier to create an approximate distribution of evolved genomes.

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. We avoid 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 11.

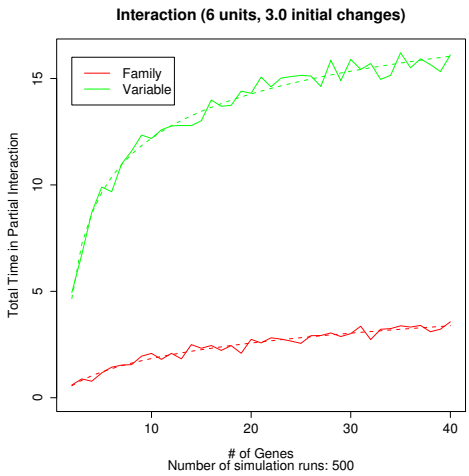

Figure 11: Simulation Results