AN ABSTRACT OF THE THESIS OF

Steven Kollmansberger for the degree of Master of Science in Computer Science presented on December 12, 2005.

Title: A Domain-Specific Embedded Language for Probabilistic Programming

Abstract approved: _____

Martin Erwig

Functional programming is concerned with referential transparency, that is, given a certain function and its parameter, that the result will always be the same. However, it seems that this is violated in applications involving uncertainty, such as rolling a dice. This thesis defines the background of probabilistic programming and domain-specific languages, and builds on these ideas to construct a domain-specific embedded language (DSEL) for probabilistic programming in a purely functional language. This DSEL is then applied in a real-world setting to develop an application in use by the Center for Gene Research at Oregon State University. The process and results of this development are discussed.

A Domain-Specific Embedded Language for Probabilistic Programming

by

Steven Kollmansberger

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 12, 2005
Commencement June 2006

Master of Science thesis of Steven Kollmansberger presented on December 12, 2005

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Steven Kollmansberger, Author

ACKNOWLEDGMENTS

This thesis would not be possible without the ideas, drive and commitment of many people. First and foremost, I thank my advisor, Martin Erwig, for shaping and molding vague ideas into reachable goals. I thank the Committee, namely, Margaret Burnett, Michael Quinn and Tim Budd, for taking time out of their busy schedules to attend multiple events and work with me throughout this process.

I would also like to acknowledgement the members of the Center for Gene Research who spent a year helping us develop the genome evolution model which provided the impetus for the probabilistic DSEL presented here. In particular, Jim Carrington, Ed Allen, Kristin Kasschau and Chris Sullivan.

A debt of gratitude must be extended to Paul Cull for his coffee hours, which provided a crucial place to unwind and bounce ideas around casually.

I appreciate the family and friends who supported me and encouraged me to hang in there; my parents, Lisa, Brent and John.

Of course, I wouldn't even be in graduate school if it wasn't for Susan Mabry, who pushed and encouraged me to apply and attend.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

LIST OF TABLES

A Domain-Specific Embedded Language for Probabilistic Programming

1. INTRODUCTION

At the heart of functional programming rests the principle of referential transparency, which in particular means that a function f applied to a value x always yields one and the same result $y = f(x)$. This principle seems to be violated when contemplating the use of functions to describe probabilistic events, such as rolling a die: It is not clear at all what exactly the outcome will be, and neither is it guaranteed that the same value will be produced repeatedly.

This thesis addresses this issue of representing uncertainty in functional languages with a domain-specific embedded language (DSEL) for performing probabilistic computation in a pure functional language (Haskell). The DSEL is applied to a real problem outside the domain of computer science.

1.1. Background

This thesis ties together the seemingly disparate concepts of domain-specific embedded languages (DSELs) and probabilistic programming. Both of these concepts are supported by a wide body of literature.

1.1.1. Domain-Specific Languages

Beyond the development of traditional, general-purpose programming languages, certain applications required the use of specialized languages. These lan-

guages came to be known as domain-specific languages (DSLs) [4]. Many domain-specific languages exist and remain popular to this day. Database queries, for example, are often performed using SQL, which is a domain-specific language for querying relational databases. Popular domain-specific languages also exist for compiler construction [33] and document formatting [38]. Many other fields, such as video driver design [64], also sport DSLs.

However, domain-specific languages have many weaknesses. Since they only operate on a small application domain, they are often used (embedded) within a larger general purpose language as strings. This prevents any checking of the DSL program before run-time. Even if the DSL supports advanced compile time checks, the compiler for the host language will see only a string, and the DSL error will not be caught until it is run.

Another difficulty comes from the need to mix features of host language and the DSL. This is often done by performing all the computation in the host language, then generating a string which represents the appropriate DSL program. Sometimes an awkward mix of host and DSL processing is used that introduces unnecessary complexity. Such constructions also introduce substantial opportunity for error. For example, in the domain-specific language SQL, strings are terminated with a single quote. If you were going to insert the string `Mary's little lamb` into the database, and just inserted it into a query string, the SQL compiler would interpret the `'` as closing the string, and an error would occur. This error could, in some cases, be used to modify the intent of the original SQL query. A whole class of exploits known as SQL injection attacks [5] take advantage of this phenomenon.

A final difficulty arises from requiring the programming to use and remember various languages. In addition to the host language, the programmer will need to be well versed on the syntax and operators of each DSL they use.

In many cases, however, it is possible to achieve the advantages of a DSL without the disadvantages. This is done by constructing the DSL out of elements of the host language, thus allowing all host language features to be used. Thus, instead of strings, the domain-specific language is represented by combinators and variables defined and typed in the host language. Languages following such an approach are called *domain-specific embedded languages* (DSELs) [32].

1.1.2. Probabilistic Programming

Early simulations used general-purpose languages, such as C or Fortran. Probabilistic computation was performed explicitly, using random numbers. Such an approach forces an undesirable coupling of problem and implementation. Later, domain-specific languages such as MATLAB were introduced which provided constructs for simulation, although still through the vein of random numbers [59].

Simulation languages, such as Simulink [14] and Psim-J [25], were developed to provide the essential components of process and object interaction to simulation designers who are not professional programmers. These systems allowed a more implicit representation of random numbers, with a focus on the actual probabilities involved in behavior.

A Bayesian DSL was introduced by Park, et al. [48, 47]. The language shown completely abstracts away the process of selecting random numbers, leaving the user to specific only probabilities. However, the authors' work is includes only

a fixed random sampling based on probabilities, and does not allow complete distribution construction.

It is not intrinsically necessary to bring in random numbers when dealing with probabilities. Various authors have shown that probability distributions can be seen as a monad [22], which is a method for encapsulating computation. In this case, a distribution is simply a list of values and their probabilities. A computation is given in the general form of a function from a value to a distribution. The monad then threads the function, applying it to each value in the distribution, and combining the results to create a new distribution [50].

By abstracting away the use of random numbers, we can present a uniform interface to the concept of probabilistic programming. A back-end system can use random numbers as needed to select elements from a distribution, or maintain an entire distribution in the monadic way.

1.2. Motivation

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 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 [41, 46, 13].

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 [52]. 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 [19].

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 [21, 12, 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 our domain-specific embedded language (DSEL) for probabilistic programming [20].

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.

1.3. Structure of this Thesis

This thesis introduces a DSEL for probabilistic programming. In Chapter 2, we describe the Probabilistic Functional Programming library in detail, and provide copious examples of its application. A significant application, a model of genome evolution, is presented in Chapter 3. Related work, both regarding biological computation and probabilistic programming, is given in Chapter 4. Finally, closing words and conclusions appear in Chapter 5.

2. PROBABILISTIC FUNCTIONAL PROGRAMMING

In this chapter, we introduce the Probabilistic Functional Programming (PFP) DSEL. The core of the PFP DSEL is a monad for probabilistic computation. We then provide combinators for combining monads and monadic values. The end user does not need to be familiar with the technical details of monads to effectively use the PFP DSEL.

This chapter opens in Section 2.1 with an overview of the two cornerstones of the DSEL, namely, distributions and transitions. Next, we "open the hood" and examine some of the technical details behind the implementation in Section 2.2. Additional functions provided by the library are shown in Section 2.3. We then break for several examples, demonstrating how the DSEL is applied in Section 2.4. Moving on, we describe how the DSEL provides randomization in Section 2.5, tracing in Section 2.6, and visualization in Section 2.7. A final example to tie all the concepts together is described in Section 2.8.

2.1. Distributions and Transitions

The probabilistic functional programming approach is based on *distributions*. A distribution represents the outcome of a probabilistic event as a collection of all possible values, tagged with their likelihood.

Distributions can represent events, such as the roll of a die or the flip of a coin. The function `uniform` turns a list of values into a distribution where each value is equally likely. We can define, for example, the outcome of die rolls.

```
die = uniform [1..6]
```

If we evaluated the value `die`, we would see all possible values and their probability. Note that the slight variance from 100% is due to display rounding.

```
> die
1  16.7%
2  16.7%
3  16.7%
4  16.7%
5  16.7%
6  16.7%
```

We would also like to be able to work with probabilistic values in a straightforward way. For this we introduce the concept of a *transition*. A transition is a function that takes a value and produces a distribution. For example, we could define a transition which, given a number, either adds one or doesn't with equal probability.

```
plus1 x = uniform [x, x+1]
```

For example, if we applied this function to the value one, we would receive:

```
> plus1 1
1  50.0%
2  50.0%
```

Creating a distribution of two elements seems to be a common task, so we created a function just for that purpose. The function `choose` takes a probability and two elements, and creates a distribution. We could redefine `plus1` using `choose`.

```
plus1 x = choose 0.5 x (x+1)
```

We can then combine these two to roll a die and then possibly add one to the result. The mechanics of handling all the various values and probabilities is dealt with automatically by the DSEL.

```
droll = do
        d <- die
          plus1 d
```

By either adding or not to each value, we make the extreme values less likely.

```
> droll
2  16.7%
3  16.7%
4  16.7%
5  16.7%
6  16.7%
1   8.3%
7   8.3%
```

2.2. The Probability Monad

This section assumes the reader is familiar with the concept of monads in functional programming. An introduction to monads is given in APPENDIX A. Distributions are constructed as a list of pairs—value and probability.

The type `Dist` is parameterized by a type variable `a`. This allows distributions of different types of elements to be constructed. In the previous section, we saw distributions of integers (`Dist Int`), however, distributions can contain values of any type. In order to differentiate distributions from arbitrary lists, we use the constructor `D`. In order to extract the list, pattern matching or a deconstruction function is required. We provide the function `unD`, shown here, which transforms a distribution back into a simple list of pairs.

```
newtype Probability = P Float
newtype Dist a = D {unD :: [(a,Probability)]}
```

This representation is shown here just for illustration; it is completely hidden from the users of the library by means of functions which construct and operate on

distributions. In particular, all functions for building distribution values enforce the constraint that the sum of all probabilities for any non-empty distribution is 1. In this way, any `Dist` value represents the complete sample space of some probabilistic event or "experiment".

In many cases, we wish to compose functions which operate on probabilistic values. A versatile approach is to construct functions of the type `a -> Dist b` and then be able to compose them. This is precisely the operation of the monadic bind function.

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

In our particular case:

```
(>>=) :: Dist a -> (a -> Dist b) -> Dist b
```

A concept that is employed in many examples is the notion of a *probabilistic function*, that is, a function that maps values into distributions. For example, the second argument of the bind operation is such a probabilistic function. Since it turns out that in many cases the argument and the result type are the same, we also introduce the derived notion of a *transition* that is a probabilistic function on just one type.

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

We can thus make `Dist` an instance of `Monad`.

```
instance Monad Dist where
  return x    = D [(x,1)]
  (D d) >>= f = D [(y,q*p) | (x,p) <- d, (y,q) <- unD (f x)]
  fail        = D []
```

The definition of `>>=` is based on the concept of *list comprehension*. First, each element and its probability is extracted from the first distribution by `(x,p) <-`

`d`. For each element, the transition `f` is applied (producing a distribution). Each element and probability is then extracted from this result distribution with `(y,q) <- unD (f x)`. Each resultant element is accumulated, with its probability being the probability of `x` multiplied by the probability of `y`. This process is diagrammed in Figure 2.1.

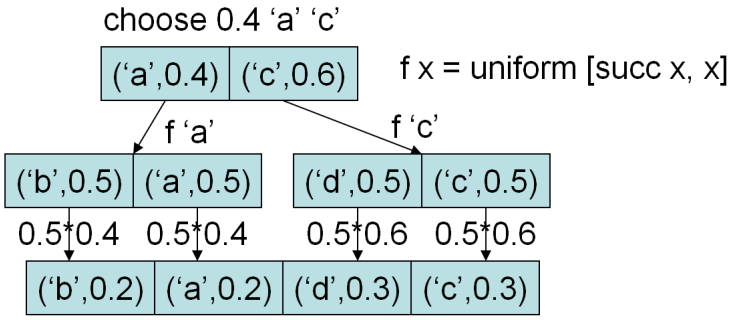

FIGURE 2.1. Composition in Action

The functions `return` and `fail` can be used to describe outcomes that are certain or impossible, respectively. We also use the synonyms `certainly` and `impossible` for these two operations. We will also need monadic composition of two functions and a list of functions.

```
(>@>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
f >@> g = (>>= g) . f

sequ :: Monad m => [a -> m a] -> a -> m a
sequ = foldl (>@>) return
```

We have also defined `Dist` as an instance of `MonadPlus`. The class `MonadPlus` provides a method for combining two monads. In this case, we can "plus" two distributions to get a 50/50 selection of the elements in the distributions. As explained previously, the function `choose` takes a probability and two elements, creating a distribution of the two elements. Since the elements are themselves

distributions, the result is a distribution of distributions. The function `unfoldD` takes such a distributions and flattens it into a single distribution. The function `mzero` produces the empty (impossible) distribution.

```
instance MonadPlus Dist where
  mzero      = D []
  mplus d d' | isZero d || isZero d' = mzero
             | otherwise             = unfoldD $ choose 0.5 d d'
```

In many cases, we want to transform the elements of a distribution without modifying the probabilities. The implementation reveals that `Dist` is also a functor. We will also use the function name `mapD` to refer to `fmap` to emphasize that the mapping is across distributions.

```
instance Functor Dist where
  fmap f (D d) = D [(f x,p) | (x,p) <- d]

mapD :: (a -> b) -> Dist a -> Dist b
mapD = fmap
```

The observation that probability distributions form a monad is not new [27]. However, previous work was mainly concerned with extending languages by offering probabilistic expressions as primitives and defining suitable semantics [35, 44, 50, 48]. The focus of those works is on identifying semantics to support particular aspects, such as the efficient evaluation of expectation queries in [50] by using a monad of probability measures or covering continuous distributions in addition to discrete ones by using sampling functions as a semantics basis [48] (and sacrificing the ability to express expectation queries). However, we are not aware of any work that is concerned with the design of a probability and simulation library based on this concept.

2.3. Probabilistic Functions

The library provides functions for creating and managing distributions, as well as extracting probabilities and values from them. We can construct distributions from lists of values using *spread* functions, that is, functions of the following type.

```
type Spread a = [a] -> Dist a
```

The library defines spread functions for various well-known probability distributions, such as `uniform` or `normal`, and also a function `enum` that allows users to attach specific probabilities to values. Probabilities can be extracted from distributions through the function `??` that is parameterized by an *event*, which is represented as a predicate on values in the distribution.

```
type Event a = a -> Bool

(??) :: Event a -> Dist a -> Probability
(??) p = P . sum . map snd . filter (p . fst) . unD
```

For example, consider the probability of rolling greater than a three on a six-sided die. We can represent this event easily using `??`.

```
g3 = (>3) ?? die
```

Since half of the values are greater than three, we get the expected result if we evaluate this expression.

```
> g3
 50.0%
```

In contrast to events, we can combine independent distributions to obtain all possible combinations of values while multiplying their corresponding probabilities. Note that the actual combination of all values and multiplication of probabilities

is handled automatically by the monad. For efficiency reasons, we can perform normalization (aggregation of multiple occurrences of a value). The normalization function is mentioned later.

```
joinWith :: (a -> b -> c) -> Dist a -> Dist b -> Dist c
joinWith f d d' = do
        x <- d
        y <- d'
        return (f x y)

prod :: Dist a -> Dist b -> Dist (a,b)
prod = joinWith (,)
```

Examples of combined independent events are rolling a number of dice. The function `certainly` constructs a distribution of one element with probability 1.

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

Having defined distributions as monads allows us to define functions to repeatedly select elements from a collection without putting them back, which causes later selections to be dependent on earlier ones. First, we define two functions that, in addition to the selected element, also return the collection without that element.

```
selectOne :: Eq a => [a] -> Dist (a,[a])
selectOne c = uniform [(v,List.delete v c) | v <- c]

selectMany :: Eq a => Int -> [a] -> Dist ([a],[a])
selectMany 0 c = return ([],c)
selectMany n c = do (x,c1)  <- selectOne c
                    (xs,c2) <- selectMany (n-1) c1
                    return (x:xs,c2)
```

With `mapD` we can now define the functions for repeatedly selecting elements from a collection. Note that the function `fst` is used in `select` because `selectMany` returns a tuple containing the list of selected elements and the list of remaining

(unselected) elements. We wish to discard the latter. We reverse the returned list because the elements retrieved in `selectMany` are successively cons'ed onto the result list, which causes the first selected element to be the last in that list.

```
select :: Eq a => Int -> [a] -> Dist [a]
select n = mapD (reverse . fst) . selectMany n
```

With this initial set of functions we can already approach many problems found in textbooks on probability and statistics and solve them by defining and applying probabilistic functions. For example, what is the probability of getting at least 2 sixes when throwing 4 dice? We can compute the answer through the following expression.

```
> ((>=2) . length . filter (==6)) ?? dice 4
 13.2%
```

Another class of frequently found problems is exemplified by "What is the probability of drawing a red, green, and blue marble (in this order) from a jar containing two red, two green, and one blue marble without putting them back?". With an enumeration type for marbles containing the constructors `R`, `G`, and `B`, we can compute the answer as follows.

```
> (==[R,G,B]) ?? select 3 [R,R,G,G,B]
  6.7%
```

2.4. Two Examples

To further illustrate the use of the concepts defined so far, we discuss two larger examples in this section: The Monty Hall Problem, and a biological example of Tree Growth.

2.4.1. The Monty Hall Problem

In the Monty Hall problem, a game show contestant is presented with three doors, one of which hides a prize. The player chooses one of the doors, and then the host opens another door which does not have the prize behind it. The player then has the option of staying with the door they have chosen or switching to the other closed door. This problem is given a theoretical probabilistic treatment in [44], and another approach to representing the problem in a probabilistic language is shown in [31].

When presented with this problem, most people will assume that switching makes no difference—since the host has opened a door without the prize, it leaves a 50/50 chance between the remaining two doors.

However, statistical analysis has shown that the player doubles their chance of winning if they switch doors. How can this be? We can use our library to determine if this analysis is correct, and how.

A simple approach is to first consider that of the three doors, only one is the winning door. Thus, the player's initial pick has a one third chance of being the winning door.

```
data Outcome = Win | Lose

firstChoice :: Dist Outcome
firstChoice = uniform [Win,Lose,Lose]
```

If the player has chosen a winning door, and then switches, they will lose. However, if they initially chose a losing door, the host only has one choice for a door to open: the other losing door. Thus, if they switch, they win. This process can be captured by a transition on outcomes.

```
switch :: Trans Outcome
switch Win  = certainly Lose
switch Lose = certainly Win
```

We can analyze the probabilities of winning by comparing `firstChoice` and applying the transition `switch` to `firstChoice`.

```
> firstChoice
Lose  66.7%
 Win  33.3%

> firstChoice >>= switch
 Win  66.7%
Lose  33.3%
```

Therefore, not switching gives the obvious one third chance of winning, while switching gives a two thirds chance of winning.

We can also model the game in more detail, replicating each step with the precise rules that accompany them. We first construct the structure of the simulation from the bottom up. We start with three doors.

```
data Door = A | B | C

doors :: [Door]
doors = [A .. C]
```

Next we create a data structure to represent the state of the game by having fields that indicate which door (`A`, `B`, or `C`) contains the prize, which is chosen, and which is opened.

```
data State = Doors {prize :: Door, chosen :: Door,
                            opened :: Door}
```

Of course, these will not all be assigned at once, but in sequence. In the initial state, the prize has not yet been hidden, no door has been chosen, and no door is open. Since the state will be evaluated only after all fields are set, we can initialize all fields with `undefined`.

```
start :: State
start = Doors {prize=u,chosen=u,opened=u} where u=undefined
```

Now each step of the game can be modeled as a transition on `State`. First, the host will choose one of the doors at random to hide the prize behind.

```
hide :: Trans State
hide s = uniform [s{prize=d} | d <- doors]
```

The function `hide` uses another example of a list comprehension. In this case, we consider each possible door as possibly hiding the prize by accumulating each door `d` from the list `doors`. A transition takes a value of some type and produces a distribution of that type. In this case, the transition `hide` takes a `State` and produces a uniform distribution of states—one state for each door the prize could be hidden behind. Next, the contestant will choose, again at random, one of the doors.

```
choose :: Trans State
choose s = uniform [s{chosen=d} | d <- doors]
```

Once the contestant has chosen a door, the host will then open a door that is not the one chosen by the contestant and is not hiding the prize. This is the first transition which depends on the value of `State` it receives by considering the value of `s` in the definition.

```
open :: Trans State
open s = uniform [s{opened=d} |
                        d <- doors \\ [prize s,chosen s]]
```

In this case, we first use the function `\\` to remove from the list of doors the door hiding the prize and the door that has been selected. Any of the remaining doors may then be opened. Next the player can switch or stay with the door already chosen. Both strategies can be represented as transitions on `State`.

```
type Strategy = Trans State
```

Switching means to chose a door that is currently not chosen and that has not been opened.

```
switch :: Strategy
switch s = uniform [s{chosen=d} |
                            d <- doors \\ [chosen s,opened s]]
```

We can also create a strategy for stay, which would simply be to leave everything precisely as it already is.

```
stay :: Strategy
stay = certainlyT id
```

For constructing transitions which produce a distribution of only one element, we provide the function `certainlyT` which converts any function of type `a -> a` into a function of type `a -> Dist a`.

```
certainlyT :: (a -> a) -> Trans a
certainlyT f = certainly . f
```

Finally, we define an ordered list of transitions that represents the game: hiding the prize, choosing a door, opening a door, and then applying a strategy.

```
game :: Strategy -> Trans State
game s = sequ [hide,choose,open,s]
```

Recall that `sequ` implements the composition of a list of monadic functions, which are transitions in this example.

If, once all the transitions have been applied, the chosen door is the same as the prize door, the contestant wins.

```
result :: State -> Outcome
result s = if chosen s==prize s then Win else Lose
```

We can define a function `eval` that plays the game for a given strategy and computes the outcome for all possible resulting states.

```
eval :: Strategy -> Dist Outcome
eval s = mapD result (game s start)
```

Again, we can determine the value of both strategies by computing a distribution.

```
> eval stay
Lose  66.7%
 Win  33.3%

> eval switch
 Win  66.7%
Lose  33.3%
```

Note that the presented model can be easily extended to four (or more) doors. All we have to do is add a `D` constructor to the definition of `Door` and change `C` to `D` in the definition of `doors`.

2.4.2. Tree Growth

Consider the simple example of tree growth. Assume a tree can grow between one and five feet in height every year. Also assume that it is possible, although less likely, that a tree could fall down in a storm or be hit by lightning, which we assume would kill it standing. How can this be represented using probabilistic functions?

We can create a data type to represent the state of the tree and, if applicable, its height.

```
type Height = Int
data Tree = Alive Height | Hit Height | Fallen
```

We can then construct a transition function for each state that the tree could be in. When the tree is alive, it grows between 1 and 5 feet every year. We distribute these values on a normal curve to make the extreme values less likely.

```
grow :: Trans Tree
grow (Alive h) = normal [Alive k | k <- [h+1..h+5]]
```

When the tree is hit, it retains its height, but when fallen, the height is discarded.

```
hit :: Trans Tree
hit (Alive h) = certainly (Hit h)

fall :: Trans Tree
fall _ = certainly Fallen
```

We can combine these three transitions into one transition that probabilistically selects which action should happen to a live tree.

```
evolve :: Trans Tree
evolve t@(Alive _) = unfoldT (enum [0.9,0.04,0.06]
                                                [grow,hit,fall]) t
evolve t           = certainly t
```

Here we use the function `enum` to create a custom spread with the given probabilities. We apply this spread to the list of transitions (`grow`, `hit`, and `fall`) which creates a distribution of transitions. The function `unfoldT` converts a distribution of transitions into a regular transition.

```
unfoldT :: Dist (Trans a) -> Trans a
```

This transition is then applied to `t`, the state of the tree, to produce a final distribution for that year. With an initial value, such as `seed = Alive 0`, we can now run simulations of the tree model.

To find out the situation after several generations, it is convenient to have a combinator that can iterate a transition a given number of times or while a certain

condition holds. Three such combinators are collected in the class `Iterate`, which allows the overloading of the iterators for transitions and randomized changes (see next section).

```
class Iterate c where
  (*.)  :: Int -> (a -> c a) -> (a -> c a)
  while :: (a -> Bool) -> (a -> c a) -> (a -> c a)
  until :: (a -> Bool) -> (a -> c a) -> (a -> c a)
  until p = while (not . p)
```

For example, to compute the distribution of possible tree values after `n` years, we can define the following function.

```
tree :: Int -> Tree -> Dist Tree
tree n = n *. evolve
```

For example, the range of possibilities after two years of growth are substantial—the tree could be killed right away, or have grown up to ten feet.

```
> tree 2 seed
 Alive 6  16.3%
 Alive 7  15.0%
 Alive 5  12.8%
 Alive 8  11.7%
  Fallen  11.4%
 Alive 4   8.5%
 Alive 9   7.4%
 Alive 3   4.6%
   Hit 0   4.0%
Alive 10   3.1%
 Alive 2   1.7%
   Hit 3   0.8%
   Hit 4   0.8%
   Hit 2   0.7%
   Hit 5   0.7%
   Hit 1   0.5%
```

We could also consider following the tree until it either died, or grew beyond five feet. We use the `until` function to continually apply the transition `evolve` until the tree is dead or has grown beyond five feet.

```
done (Alive x) = x >= 5
done _ = True

ev5 = until done evolve
```

We find it is still most likely that it will be alive and growing.

```
> ev5 seed
Alive 5  32.9%
Alive 6  17.0%
Alive 7  15.1%
 Fallen  11.3%
Alive 8  10.8%
Alive 9   5.4%
  Hit 0   4.0%
  Hit 4   1.2%
  Hit 3   1.0%
  Hit 2   0.8%
  Hit 1   0.5%
```

For large values of `n`, computing complete distributions is computationally infeasible. In such cases, randomization of values from distributions provides a way to approximate the final distribution with varying degrees of precision.

2.5. Randomization

The need for randomization arises when each transition creates a new set of values for each value currently in the distribution, thus creating an exponential space explosion. We provide functions to transform a regular transition into a *randomized change*, which selects only one result from the created distribution.

Library users need not design transitions to be randomized. Instead, transitions can be created once, then automatically randomized as needed by employing corresponding library functions.

All randomized values live within the `R` monad, which is simply a synonym for `IO`. Elementary functions to support randomization are `pick`, which allows the selection of exactly one value from a distribution, randomly yielding one of the values according to the specified probabilities, and `random` which transforms a transition into what we call a *randomized change*.

```
type R a = IO a

pick :: Dist a -> R a
pick d = Random.randomRIO (0,1) >>= return . selectP d
```

For the function `pick`, the distribution is considered to span the range zero to one, with each element having a width equal to its probability. In this case, although the numbers generated by the random number generator are all equally likely, they are more likely to fall on "larger" elements (those with higher probabilities). We can see the use of `pick` in selecting one value from the die roll. We need the function `printR` to extract the result from the monad and display it.

```
> printR $ pick die
1
> printR $ pick die
6
> printR $ pick die
3
> printR $ pick die
2
> printR $ pick die
5
```

We can compose `pick` with a transition to create a randomized change.

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

random :: Trans a -> RChange a
random t = pick . t
```

Randomly picking a value from a distribution or randomizing a transition is not an end in itself. By repeatedly applying such a randomized change to the same value, we can construct an arbitrarily good approximation of the exact probabilistic distribution. This is known as Monte Carlo sampling [55]. The collection of values obtained by repeated application of randomized changes can be aggregated to yield an approximation of a distribution, represented by the type `RDist a`. Given a list of random values, we can first transform them into a list of values within the `R` monad. Then we can assign equal probabilities with `uniform` and group equal values by a function `norm` that also sums their probabilities. This process is shown in Figure 2.2.

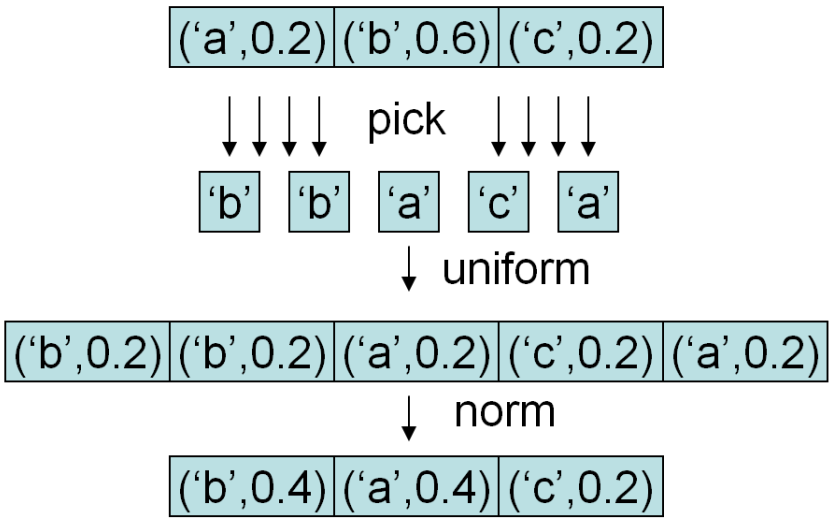

FIGURE 2.2. Creating a Randomized Distribution

```
type RDist a = R (Dist a)

rDist :: Ord a => [R a] -> RDist a
rDist = fmap (norm . uniform) . sequence
```

For example, consider rolling a die three times and using those values to build an approximate distribution. We can do this with `pick` and `rDist`.

```
> printR $ rDist [pick die, pick die, pick die]
1  33.3%
3  33.3%
4  33.3%

> printR $ rDist [pick die, pick die, pick die]
3  66.7%
6  33.3%
```

Of course, with only three samples the approximation is very crude. However, by increasing the sample size we can improve the precision of the approximation. The function `replicate` produces a list containing n copies of the given argument (20 and 100, in these cases).

```
> printR $ rDist (replicate 20 (pick die))
1  25.0%
6  25.0%
5  20.0%
4  15.0%
3  10.0%
2   5.0%

> printR $ rDist (replicate 100 (pick die))
5  19.0%
4  18.0%
1  17.0%
6  17.0%
2  16.0%
3  13.0%
```

With `rDist` we can implement a function `~.` that repeatedly applies a randomized change or a transition and derives a randomized distribution. The `Ord` constraint

on `a` in the signature of `~.` is required because the instance definitions are based on `norm` (through `rDist`).[1]

```
class Sim c where
  (~.) :: Ord a => Int -> (a -> c a) -> RTrans a
  ...

instance Sim IO where
  (~.) n t = rDist . replicate n . t
  ...

instance Sim Dist where
  (~.) n = (~.) n . random
  ...
```

In particular, the latter instance definition allows us to simulate transitions in retrospect. In other words, we can define functions to compute full distributions and can later turn them into computations for randomized distributions without changing their definition. For example, the tree growth computation that is given by the function `tree` can be turned into a simulation that runs a randomized tree growth `k` times as follows.

```
simTree :: Int -> Int -> Tree -> RDist Tree
simTree k n = k ~. tree n
```

An example of the space explosion compared to the roughly constant space requirement in randomization is shown in Table 2.1 for the tree growth simulation.

Similarly, for the Monty Hall problem we could randomly perform the trial many times instead of deterministically calculating the outcomes.

---

[1]The function `norm` sorts the values in a distribution to achieve grouping for efficiency reasons. Since we have found that in most examples that we encountered defining an `Ord` instance is not more difficult than an `Eq` instance, we have preferred the more efficient over the more general definition.

TABLE 2.1. Comparing the maximum heap size (in kilobytes) for fully simulated and randomized tree growth simulations

Generations	Fully simulated	Randomized (500 runs)
5	700	650
6	4000	800
7	19000	800

```
simEval :: Int -> Strategy -> RDist Outcome
simEval k s = mapD result `fmap` (k ~. game s) start
```

In the function `simEval`, we simulate the game `k` times with the given strategy, map the `result` function (which determines win or loss) and accumulate the results. With a reasonably high number of runs, the resultant distribution should closely approximate the actual distribution.

```
> printR $ simEval 1000 stay
Lose  65.8%
 Win  34.2%

> printR $ simEval 1000 switch
 Win  68.3%
Lose  31.7%
```

Since in many simulation examples it is required to simulate the `n`-fold repetition of a transition `k` times, we also introduce a combination of the functions `*.` and `~.` that performs both steps. We add this function to the `Sim` class.

```
class Sim c where
  ...
  (~*.) :: Ord a => (Int,Int) -> (a -> c a) -> RTrans a

instance Sim IO where
  ...
  (~*.) (k,n) t = k ~. n *. t

instance Sim Dist where
  ...
  (~*.) x = (~*.) x . random
```

Note that `*.` is defined to bind stronger than the `~.` function. We can thus implement the tree simulation also directly based on `evolve`.

```
simTree k n = (k,n) ~*. evolve
```

Again, we do not have to mention random number generation anywhere in the model of the application.

2.6. 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.

To facilitate tracing, we define types and function to produce traces of probabilistic and randomized computations. For deterministic and probabilistic values we introduce the following types.

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

A walk is a function that produces a trace, that is, a list of values. Continuing the idea of iteration described in the previous section, we define a function to generate walks, which is simply a bounded version of the predefined function `iterate`, which creates an infinite list by applying a function over and over again to the previous value.

```
walk :: Int -> Change a -> Walk a
walk n f = take n . iterate f
```

Note that the type `Change a` is simply a synonym for `a -> a`, introduced for completeness and symmetry (see `RChange a` above).

While a walk produces a trace, iteration of a transition yields a list of distributions, which represents the explored probability space. We use the symbol `*..` to represent the trace-producing iteration. The definition is based on the function `>>:`, which prepends the result of a transition to a space. Since a transition is a function, we also return a function which returns a space, namely an `Expand`.

```
(*..) :: Int -> Trans a -> Expand a
0 *.. _ = singleton . certainly
1 *.. t = singleton . t
n *.. t = t >>: (n-1) *.. t
```

The potential space problem of "simple" iterations is even more so present in trace-producing iterations. Therefore, we also define randomized versions of the types and iterators.

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

The function `rWalk` iterates a random change to create a random walk, which can produce a random trace.

```
rWalk :: Int -> RChange a -> RWalk a
```

The definition is similar to that of `*..`, but not identical, because the result types are structurally different: While `Dist` is nested within `Trace`, `R` is wrapped around `Trace`. This is also the reason why we cannot overload the notation for these two functions.

Similar to `~.` we can now implement a function `~..` that simulates the repeated application of a randomized change or transition and derives a randomized space, that is, a randomized sequence of distributions that approximate the exact distributions obtained during tracing. Since `~..` is overloaded like `~.` for transitions and random changes, it can reside in the same class `Sim`. In the second instance definition, `mergeTraces` transposes a list of random lists into a randomized list of distributions, which represent an approximation of the explored probabilistic space.

```
class Sim c where
  ...
  (~..) :: Ord a => (Int,Int) -> (a -> c a) -> RExpand a

instance Sim IO where
  ...
  (~..) (k,n) t = mergeTraces . replicate k . rWalk n t

instance Sim Dist where
  ...
  (~..) x = (~..) x . random
```

Note that the first argument of `~..` is a pair of integers representing the number of simulation runs *and* the number of repeated application of the argument function. The latter is required to build the correct number of elements in the trace unlike for `~.` where only the final result matters.

Applied to the tree-growth example we can now define functions for computing an exact and approximated history of the probabilistic tree space as follows.

```
hist :: Int -> Tree -> Space Tree
hist n = n *.. evolve

simHist :: Int -> Int -> Tree -> RSpace Tree
simHist k n = (k,n) ~.. evolve
```

In summary, we have constructed five main operators which transform a transition into a generator, either random or complete, single-valued or trace. The layout of the four basic operators is shown in Table 2.2. The fifth operator is `~*.`, which, as previously mentioned, is simply the combination of `~.` and `*..`

TABLE 2.2. Four Basic Iteration Operators and Their Result Types

	`*` (complete)	`~` (randomized)
`.` (single)	`Dist`	`RDist`
`..` (trace)	`Space`	`RSpace`

2.7. Visualization

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. Our library provides

graphing functionality through the statistical package R. We provide access to a number of R graphical customizations to make the resultant graphs both useful and aesthetic.

Consider again the tree example. We can find the distribution of tree heights after five years with the expression `tree 5 seed`. However, this distribution consists of 43 different entries, a mix of living trees, fallen trees, and trees hit by lightning. We can build a function which extracts the height of a given tree.

```
height :: Tree -> Int
height Fallen = 0
height (Hit h) = h
height (Alive h) = h
```

A distribution of values can be turned into a plot using `plotD`. The function `plotD` plots the values on the X-axis and the associated probabilities on the Y-axis. One or more plots can be transcribed into an R data file using the function `fig`. Note that `Vis` is a synonym for `IO ()`, the IO monad.

```
plotD :: ToFloat a => Dist a -> Plot
fig :: [Plot] -> Vis
```

With these functions, we can graph the distribution of tree height at year five with a straight-forward expression, which produces the graph shown in Figure 2.3.

```
year5 = fig [plotD $ mapD height (tree 5 seed)]
```

This graph is very plain. It would be helpful to have some annotations to indicate what `x` and `f(x)` represent. We allow optional annotation for title and axes. This is done using the `figP` function which takes figure annotation as well as a list of plots. In order to allow the user to specify only those annotation of interest, and so that the annotation may be expanded without breaking existing applications, annotations are specified as changes from the default figure. The result is shown in Figure 2.4.

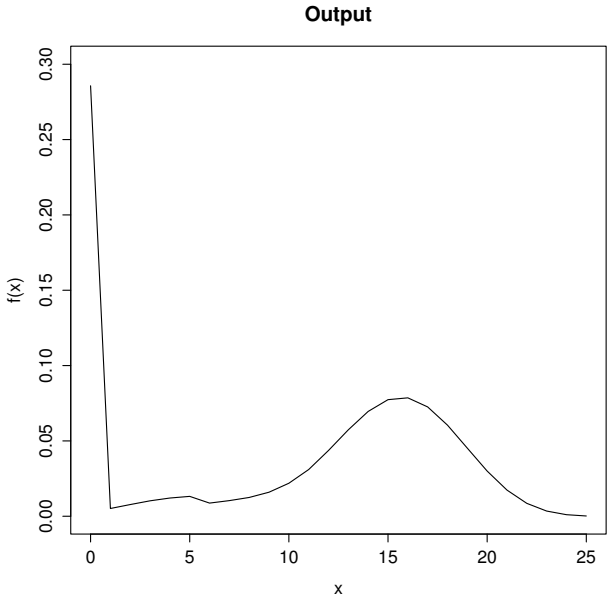

FIGURE 2.3. Tree Height at Five Years

```
year5 = figP figure{title="Tree Growth",
            xLabel="Height (ft)",
            yLabel="Probability"}
            [plotD $ mapD height (tree 5 seed)]
```

What if we wanted to see the tree's growth at not only five years, but also three and seven for comparison purposes? In that case, we can save some space by defining a function which creates a plot for a given year.

```
heightAtTime :: Int -> Plot
heightAtTime y = plotD $ mapD height (tree y seed)
```

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

```
years = figP figure{title="Tree Growth",
            xLabel="Height (ft)",
            yLabel="Probability"}
            [heightAtTime 3, heightAtTime 5,heightAtTime 7]
```

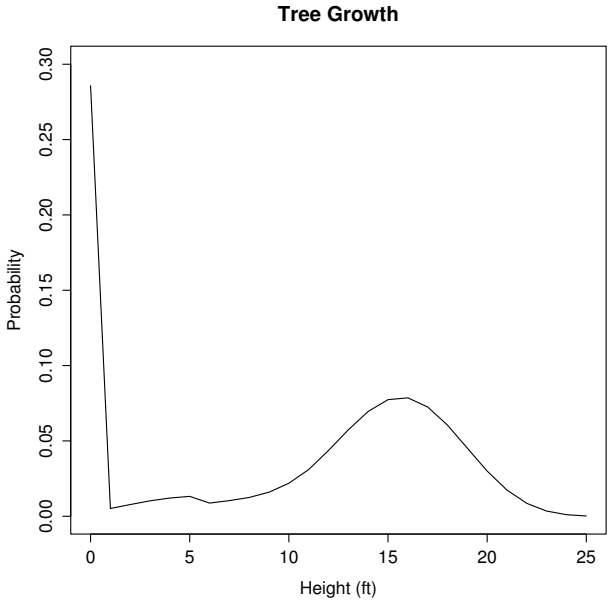

FIGURE 2.4. Tree Height at Five Years, with Labels

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. The resultant graph is shown in Figure 2.5.

```
years = figP figure{title="Tree Growth",
            xLabel="Height (ft)",
            yLabel="Probability"}
            (map heightAtTime [3,5,7])
```

Perhaps we can determine which curve refers to which year, but this may not be obvious to all viewers. It would be nice if there were a straightforward method for providing a legend or other annotation of each curve. Our system allows each

FIGURE 2.5. Tree Height at Three, Five and Seven Years

curve to be colored and a legend to be provided. By providing one or more labels, a legend will automatically be created.

```
years = figP figure{title="Tree Growth",
            xLabel="Height (ft)",
            yLabel="Probability"}
            [(heightAtTime 3){color=Blue,label="3 Years"}
            (heightAtTime 5){color=Green,label="5 Years"}
            (heightAtTime 7){color=Red,label="7 Years"}]
```

As before, we can define a function which will create a plot with for a given year, and assign it a label and color.

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

We use `map`, as before, to create a new graph with colors and a legend. The final graph is shown in Figure 2.6.

```
yearsc = figP figure{title="Tree Growth",
            xLabel="Height (ft)",
            yLabel="Probability"}
            (map heightCurve
            [(3,Blue),(5,Green),(7,Red)])
```

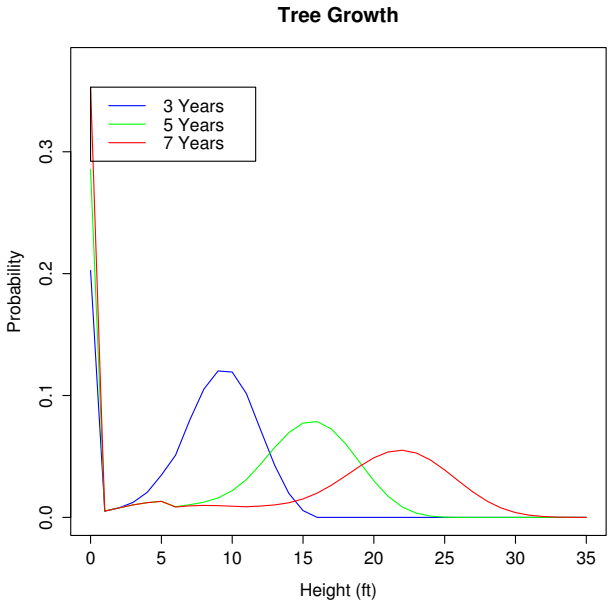

FIGURE 2.6. Tree Height, with Color and Legend, at Three, Five and Seven Years

2.8. Another Biology Example

We can bring together many aspects of the DSEL, such as randomization, simulation and visualization in one example. The Lotka-Volterra predator-prey model [40] 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). The search rate refers to how able the predators are to seek out the prey, and the energetic efficiency determines how much growth the predator population receives from consuming prey. 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 pair 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 two distributions and two constants 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` (short for delta-victim-predator, the change of each population at each step), 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. Note that the term `` `max` 0 `` is used to ensure that the prey population can never drop below zero.

```
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 explosion, 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 `n` generations long to produce a randomized space.

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

We then apply the visualization module to present 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 based on a weighted average.

We devise a function which operates on the randomized space to extract distributions of numeric values so that we can 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` (representing either `fst` or `snd`) 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''))
```

Our original idea behind the PFP DSEL was to empower biologists and other scientists to construct simulation models themselves, using straightforward syntax and easily understood concepts which closely related to their ideas of the problem. However, as the definition of `getRE` shows, there is still much work to be done before the goal of making PFP usable by end-users becomes realized.

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 on the left in Figure 2.7.

Compared with the corresponding deterministic model with `growth = 1.055` and `death = 1.95` (shown on the right in Figure 2.7), 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 [53], who notes that stochastic predator-prey models almost always experience extinction after several generations.

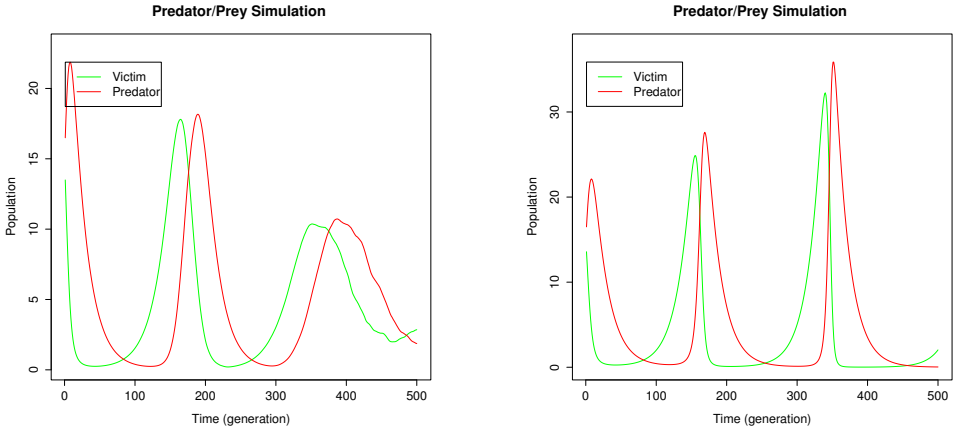

FIGURE 2.7. Probabilistic (on the left) and Deterministic (on the right) Predator/Prey Simulation over 500 Generations

3. MODELING GENOME EVOLUTION

The development of the PFP library did not occur in a vacuum. Instead, a real application need motivated the development. In conjunction with the Center for Gene Research at Oregon State University, we worked to develop a model for the evolution of microRNAs [21, 12, 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. In this chapter, we discuss the development and final version of the genome evolution application.

We open this chapter in Section 3.1 with a report on the gradual development of the genome model through iterations over several prototypes, followed by evaluations and discussions with biologists. The final genome evolution model is described in Section 3.2.

3.1. Model Prototyping

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 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. From our experience it seems 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 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 3.1. 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.

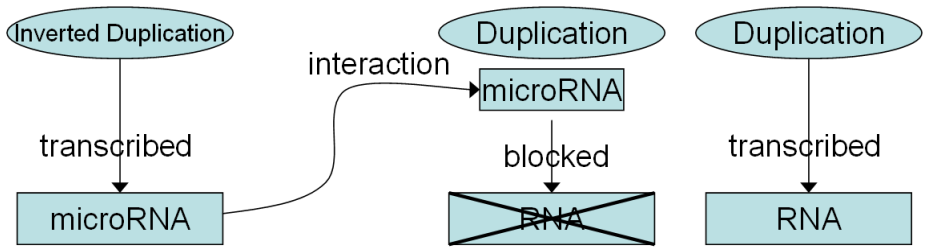

FIGURE 3.1. Effects of microRNAs on Gene Duplications

To this end we had to model a genome that accumulates changes over time. 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.

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.

```
type Dup = Int
```

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 a strand of RNA folded onto itself. This can be viewed as two strands (called sense and anti-sense) and a loop.

We then allowed a change to occur either in one of the parts of the inverted duplications 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.

```
type IDup = (Int, Int)
```

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.

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

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 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 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`.

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

interaction :: Genome -> Interaction
interaction ((s,a),gs) | s+a>5      = Loss
                       | l==0       = None
                       | l==g       = Full
                       | otherwise  = 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 3.2.

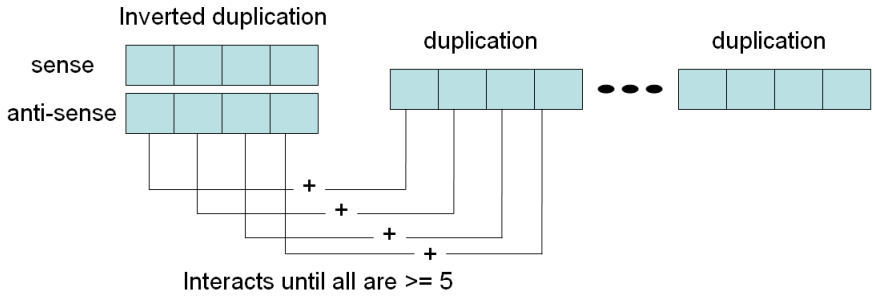

FIGURE 3.2. The Test of Interaction

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

3.2. 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. 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` $= 1/3$) 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 $(u * n)/(2u + u * n)$ which is the same as $n/(2 + n)$.

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)
```

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

```
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, 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
```

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 3.3.

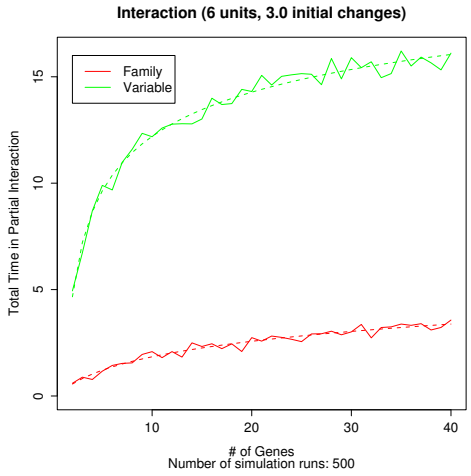

FIGURE 3.3. Simulation Results

MicroRNAs are significant in determining the function of genes. However, it is not completely clear how 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] support both of these predictions.

4. RELATED WORK

In this chapter, we present work related to this thesis. An overview of work done in simulation and control, with a particular emphasis on languages, is presented in Section 4.1. We then introduce functional and monadic probability systems, that is, other approaches to representing probabilistic computation using functional languages, in Section 4.2. Finally, we discuss other approaches to biological modeling, including mathematical, theoretical, and domain-specific languages in Section 4.3.

4.1. Simulation and Control

A popular use of computing technology has long been simulation and control. In this field, computers simulate real world events, or control external hardware processes. Modern control systems are often constructed with LabVIEW, a visual dataflow programming language. LabVIEW is programmed with a diagram structure that approximates wiring diagrams, while the user interface to the designed application appears like an instrument panel. In this way, LabVIEW is designed so that applications may be both built and used by scientists and engineers [65]. LabVIEW allows for sophisticated use of control structures, data processing and random numbers, but is not explicitly designed for simulation purposes [7].

A number of platforms and toolkits do exist primarily for simulation purposes. Many of these depend on object-oriented programming languages, assuming that objects are a good way to represent factors of a simulation. One such

platform is the High Level Architecture (HLA) introduced by Kuhl et al. which builds on the Java programming language [39]. The HLA is primarily concerned with large simulations which are constructed by a heterogeneous team of people. Randomization and probability are not primary concerns, whereas time management and object interaction are considered the key points of an HLA simulation.

A general overview of simulation is provided by Robinson [54]. He describes simulations which progress over time using discrete events, time slices, or continuous simulation. The author describes the use of random numbers to derive distributions appropriate for simulating uncertainty in a model, whereas our system abstracts the random numbers away. He also indicates that visual simulation software is by far the preferred method of designing and executing simulations, as it allows the user a visual, and thus more directly informative, understanding of what is happening. A smaller amount of simulation is done in spreadsheets or traditional programming languages.

Fishman provides a detailed description of the methods and statistics for discrete event simulation [23]. He discusses several platforms for modeling simulations, including SIMSCRIPT, SIMAN and LABATCH. The author also describes methods for analyzing output from various starting parameters to determine statistically significant information from simulations. In addition, a large section of the text is devoted to describing the precise algorithmic method for deriving a sample from a wide range of standard probability distributions. These sampling functions operate on a uniform random number generator. Our system allows such distributions to be constructed explicitly without worrying about the random numbers.

Pooch and Wall [49] provide an overview of the mathematical concepts of simulation, with discussion of the computational elements as well. In particular,

the authors discuss distributed simulation and queueing theory. Distributed simulation occurs when a single simulation is broken in synchronized components and executed independently. Since many simulations involve queues, the authors make special mention of the techniques involved in simulated first-in-first-out queues.

A theoretical treatment of the mathematical foundations of simulation is provided by Banks et al. in [3]. The authors cover sources of randomness in simulations, as well as approaches to modeling input and output, and verifying a simulation's accuracy. They also offer comparisons of various mathematical approaches to modeling certain classes of scenarios.

Garrido introduces Psim-J, a Java based simulation platform [25]. He presents several examples, including a car-wash simulation and a part replacement model, to show how the features of Java can be used to support simulations. The author spends a reasonable amount of space discussing concurrency issues in simulation—cooperation, conditional waiting, interrupts, and other similar concerns. The author does not spend much time addressing probabilistic concerns.

A Pascal simulation library is presented in [8]. This library provides a layer of functionality on top of Pascal, including interpolation functions, timing functions, input generators (for example, generating waves or pulses), and numerical integration functions. The author demonstrates the use of the library for two particular applications: a swinging pendulum and a fishery simulation. The fishery simulation involves a varying number of fishing boats on a lake where fish reproduce with a given frequency. The goal is to optimize fishing behavior for maximum sustainable catch.

MATLAB is a mathematics package that can be used to perform simulations [59]. In addition to regular imperative programming features, MATLAB offers matrix functions and a variety of random number generators. MATLAB

can also perform symbolic analysis, such as function simplification or symbolic integration. Plotting tools for both 2-D and 3-D plots are offered as well.

Methods for reducing problems to differential equations are given by Davis [18]. He introduces the mathematical modeling of multi-dimensional systems, and shows how to use MATLAB to solve the numerical equations such problems present. The text is highly focused on deriving mathematical models of problems by representing them in terms of well known problems, such as oscillation and diffusion.

Simulink [14] is a visual programming language for numerical simulations. Algorithms are described in a visual data-flow manner using various provided functions which perform numerical analysis or generate curves. More complex functions can be coded in a scripting language or in C and imported into the visual environment. The manual provides three aircraft related examples, one about the performance of an aircraft and two about autopilot design.

A wide variety of applications are modeled using the Stella language by Hannon and Ruth [29]. The models are primarily represented in terms of difference equations, although randomness is introduced as a factor in several examples, including one of natural selection. A variety of competitive models are introduced, including fisheries and tree cutting. As before, the models attempt to define the optimal point of resource extraction without destroying growth potential. The authors introduce an auto assembly example with which they demonstrate the feature called "ovens". This feature of the language automatically holds a specified finite amount of a given component for some amount of time. This feature simulates, for example, one station in an assembly line where several workers may take some time to paint a part and can only paint so many parts at a time.

In order to transform a system into a simulation, it is important to determine which components are important and which are not. It is then necessary to determine how the important components behave, either deterministically or stochastically, with equations or step by step conditional behaviors. Only once these have been determined can a model be programmed in any modeling or simulation language. Starfield et al. present a non-technical introduction to determining the important factors in a model without a heavy emphasis on mathematical details [61].

4.2. Functional and Monadic Probability Systems

A randomized probabilistic language is demonstrated by Park et al. [48], extending some earlier work by the same authors in [47]. Their method is based on sampling from probabilities, which can then be combined to form random results or probability distributions. The authors show a classic Bayesian network example of an alarm which can go off if there is either a burglary or an earthquake, and which may independently cause Mary, John, or both to call the police. The desired result is to determine the probability that Mary calls given that John calls (and nothing else is known). 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 probabilisitic result.

Encapsulation of probabilistic computation into monads is briefly discussed in [22]. The author discusses associating probabilities and values in a list and performing computation on them. He also mentions such concepts as randomly choosing a value from a distribution and ambiguous (uncertain) values. However,

the treatment is not well developed and does not include concepts such as tracing or simulation.

A theoretical, set-oriented treatment of probabilistic computations is given in [34]. The author points out that probabilistic computations can be considered in a monadic domain.

A monadic probability implementation is demonstrated by [50]. 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 die-rolling language is presented by Mogensen [43]. This language provides constructs for combining probabilistic collections of integers, including unbound repeation (while-loops). The generated results are displayed either as an expected value or an entire probability distribution.

Sato and Kameya [56] introduce a statistical logic learning language based on Prolog, called PRISM. This language is designed for modeling uncertainty at a high level, and can also infer parameters based on a set of given data.

The construction of statistical models and inference of Bayesian networks in Haskell is dealt with extensively by Allison [**?**]. He then shows how the method can be generalized to support various machine learning techniques, supported by Haskell type classes [**?**]. Finally, he introduces the paradigm of *inductive programming*, where a rough model and noisy data (estimations) are provided, and the system infers a complete solution [**?**]. In all three works, a number of case study examples are shown, with an emphasis on Minimal Message Length and Bayesian networks.

4.3. Biological Modeling Methods and Languages

The foundations of systems biology in modeling are described by Kitano [37]. He describes four critical components for any modeling method or language. First, it must be able to describe the structure of the systems, by representing objects such as genes and processes such as metabolism. Second, it must be able to describe the operations of the objects and processes. Third, it must have some method of representing the way systems and objects interact with other objects. Finally, the modeling method should allow for certain desired properties to be expressed.

Most biological modeling systems are numerically predictive. That is, given a model and an initial state, these models produce an indication of what the output state will be, or is likely to be. It is also possible to take raw data and construct a model which fits it [26].

However, these models may be insufficient to allow a biologist to express a complex biological hypothesis. Luca Cardelli [11] proposes that domain-specific modeling languages are appropriate to bridge the gap between biology and computation. He argues that many processes involve sequential actions which can best be represented by modeling languages. He also claims that biological systems can be abstracted in meaningful ways that strongly resemble software systems, and developing means of modeling biological processes may provide insight into complex software engineering problems. Such a language would allow the biologist to specify their model not in terms of advanced mathematical equations, but in terms of cells and interactions that they already know or hypothesize to exist.

In this section, we group the systems by approach: first discussing algebraic systems in Section 4.3.1, then graph systems in Section 4.3.2. Simulations written

in general purpose languages are covered in Section 4.3.3 and compared to domain-specific languages along with various "analogy" based approaches in Section 4.3.4. Finally, a theoretical collection of calculi for biological systems is presented in Section 4.3.5.

4.3.1. Algebraic Systems

A mathematical approach was taken by Nilsson and Fritzson with the Modelica system [46]. 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. The authors focus on a lambda decision circuit, which attempts to model the activity of phage infection in selecting either replication or integration. The model shown by the authors involves both switching circuits, as well as a segment which the authors note would require kinetic modeling. In this way, the circuit model is not a complete model of biological function, but a means of connecting numerical models together in a more abstract way.

A more detailed coverage of Modelica [24] describes its capability to model both discrete, continuous, and hybrid systems using rules, differential equations, and concurrency. The authors also describe how Modelica can be used to model non-biological processes, such as pipeline control and manufacturing processes.

Eker et al. introduce a method they called "pathway logic" [19], 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[1] and meta-analysis, which asks broadly which classes of starting states would satisfy some final criteria, thus allowing model disambiguation using actual data.

The pathway logic system is applied by Talcott et al. to the modeling of signal transduction processes involving proteins [62]. The model is concurrently operational at two levels of abstraction: at the high level, overall protein states are modeled using protein functional domains. If sufficient data is available, the lower level modeling extends the protein functional domains to include explicit molecular representations of signaling molecules. The model is then queried to determine if particular proteins can be activated in certain situations.

4.3.2. Graph Systems

Graph-based models have also been used by several authors. Vazquez et al. model protein interaction networks using graph generation algorithms [66]. They show that the generated graphs have features in common with interaction graphs derived from actual proteins. The generator algorithm is based on mathematical probabilities of duplication and divergence.

---

[1]For example, if some property P is true, do we always reach a state that satisfies property Q?

Xing et al. introduce a hidden markov inference model for identifying motif structures in sequences such as DNA [68]. They apply Bayesian and EM learning to the model to automatically derive the appropriate models, and then check them against experimental data. Fine tuning the learning methods provided a false positive rate as low as seven percent, with no false negatives.

Another graph model is presented by Bhan et al. [6]. Their model is concerned with genome regulation on the whole-genome scale, rather than individual protein interactions. However, the modeling concepts are similar. The authors discuss duplication, partial duplication, and rewiring (a kind of intentionally introduced error in duplication). They compare the derived graphs to the actual known genome regulation networks of yeast and find them statistically similar.

A similar approach is presented by Sole et al. [60]. They use a simple model of duplication with adding a link or removing a link at each stage, with given probabilities α (for adding) and δ (for removing). They show how a proteome interaction map generated with their model is similar to a yeast proteome interaction map derived from experimental data.

A more detailed model is provided by Teichmann and Babu [63]. The authors recognize differences between transcription factors and target genes for the purposes of duplication. They allow either one or both to be duplicated, and also allow loss or gain, which provides a similar change strategy to those mentioned previously. The authors' model can be used to provide various scenarios of duplication in evolution of *E. coli* and yeast.

4.3.3. Low-level Simulations

A discussion of quantitative factors in biological modeling is presented in [28]. The author discusses creating and simplifying numerical simulations, using population growth as an example. The author gives an example of a two equation Lotka-Volterra predator-prey model simulated in a page and a half of C code. An excerpt of this code is shown in Table 4.1 and compared to the corresponding PFP code for the same problem. Note that the PFP code is more compact and avoids entanglements with orthogonal concerns, such as output preparation.

The author also shows how little code could be needed to represent a simulation in an ideal language. The trade-off between micro-control and conciseness is noted, with the author indicating a preference for conciseness, saying that it gives freedom to the user to focus on and understand their problem. Several simulation specific languages are discussed, including SCoP, a 1986 simulation language using differential equations for physiological and biomedical systems, Stella, a visual modeling language using Forrester diagrams, along with a brief mention of mathematical packages and spreadsheets.

Wilson [67] points out how to create models that simulate the actual behavior step by step. Wilson's approach is very low-level. Each simulation model is written in C. The visualization engine he demonstrates is a PostScript generator also written in C. The author's simulations have the advantage of taking spatial issues into account. However, most of the examples are done in one-dimensional space. For example, the author works through two examples, one for a population of moving insects, another for an epidemic where each member may be either suspectible, exposed but not yet infectious, infectious, or recovered and immune. Once exposed to the disease by an infectious neighbor, an organism stays in each

TABLE 4.1. The Predator-Prey Model in Two Takes

Haefner's C Code (logic)	PFP Version (logic)
`while (t < maxt) {`	`dvp (v,p) = do`
`newV = V + r*V - b*V*P;`	`g <- growth`
`newP = P + b*c*P - d*P;`	`d <- death`
`V = newV;`	`return (g*v - s*v*p ‘max‘ 0,`
`P = newP;`	`d*p + e*v*p)`
`if (prntime <= t) {`	`ppt n = ((1000,n) ~.. dvp) (v0,p0)`
`tarray[toprint] = t;`	
`Varray[toprint] = V;`	
`Parray[toprint] = P;`	
`prntime += prndelt;`	
`toprint++;`	
`}`	
`t++;`	
`}`	

state for a particular amount of time, eventually returning to the suspectible state. These times are given deterministically, and no uncertainty is introduced into the model. Each model requires at least two full pages of C code.

4.3.4. Domain-Specific Analogies and Languages

An early attempt to model biological systems was done by McAdams and Shapiro [41]. 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.

Regev et al. [52] 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. An ambient within another ambient is called a child, and the ambient housing it is called the parent. The authors also describe a series of capabilities and processes which control what each ambient does. One ambient with an "entry" capability may become a child ambient of an ambient with an "accept" capability. Likewise, an ambient may leave another ambient if the child ambient has the "exit" ability and the parent ambient has the "expel" ability. Similarly, ambients may merge or separate.

Processes describe the way ambient functionality is coordinated. For example, two ambients may exist and operate simultaneously. A single ambient may "replicate", and exist any number of times. Messages can also be sent between ambients, which serve to instantiate behavior.

The authors describe the isolation of an ambient as being similar to the membranes of a cell—most processes within the cell are independent of the processes outside the cell except for a few processes which cross the membrane. 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, which includes the concept of ambients and is designed for systems biology simulations is briefly described.

A graphical summary of this model is presented by Cardelli in [9]. This model uses graphical rewrite rules for expressing molecular reactions. The model

is primarily composed of concurrent token generation and consumption. When a set of matching rules is found with the same name, where one produces a token and another consumes it, then the processing proceeds. The generating rule is specified as $n!\{m\}(P)$, while the consuming rule is specified as $n?\{p\}(Q)$. Processing proceeds when the names n match, and one rule is producing (!) and one is consuming (?). In this case, the first rule produces the token m, and is replaced with the molecule P, while the second rule received the token m, and is replaced by the molecule $Q\{m/p\}$, where all cases of m are replaced by the received token p.

Harrison and Harrison [30] introduce a biological modeling language that consists of cells which perform bacterial actions, such as tumbling (changing direction), growing, dividing, and dieing. The language uses Markov chains, a non-deterministic state approach wherein the state of the cell and environment determine the probability of each action that may be taken.

Antoniotti et al. describe the important ability to query a biological modeling system to determine facts about the model [2]. The authors note that such a system would work well with biologists, who often have data and a mathematical model, and want to pose queries to it. A model could be queried with formal representations of questions such as "Will the system reach a steady state?" or, more generally, "Will some event P cause another event Q?". The foundation of the authors' model is the XS system, which is a series of algebraic constraints extended with automata. Once a biological model has been expressed in these precise mathematical constructs, the biologist may query the model using a language that allows constraints to be set on various parameters and determine if certain states will occur under the given conditions.

Kam et al. employ the mechanisms of live sequence charts and play-in/play-out methods to model cell activity [36]. Live sequence charts are a visual method for describing a variety of sequenced scenarios which are certain, possible, or impossible (the latter are known as anti-scenarios). Play-in/play-out is an end-user device for constructing live sequence charts. The mechanism allows users to demonstrate behavior to the system by performing it (play-in) and then allow the system to infer the desired operational rules, which can then be run to continue the user's activity (play-out). The system automatically detects attempts at creating contradictory rules, thus keeping the model consistent. As an additional predictive feature, the authors discuss "smart play-out", which allows the user to specify an end result and determine if it could be reached from any consistent starting state in the model.

Pathway Modeling Language (PML) is introduced by Chang and Sridharan [13]. 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 comparmentalization of reactions. The authors devise a method to translate PML into π-calculus as well. 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.

Shannon et al. have introduced a data mining and visualization tool for systems biology called Cytoscape [58]. Cytoscape is designed to process large amounts of raw data, and visualize it in network form. The system allows queries on the network and arbitrary extension via plug-ins. These plug-ins, which include

modules such as genetic regulatory analysis and a protein analyzer with respect to DNA damage recovery, contain the primary functional component of Cytoscape.

4.3.5. Biological Calculi

In addition to systems which model some subset of biological functionality, it is important to devise mathematical abstractions of biological systems [51]. This allows a broad analysis and extrapolation of biological concepts, leading to the formation of hypotheses.

In particular, a good abstraction will capture the essence of the domain in a computable, comprehensible, and extensible representation. Systems biology has not developed an agreed abstract representation, although the π-calculus (a model for communication over channels) [42] is being used as a basis by many approaches. In particular, it is important to separate the implementation (live cells, simulation, etc) from the specification (the "logic" of the system) before such an implementation-independent representation can be agreed upon.

An early attempt at this formalization was done by Nagasaki et al. in their construction of Bio-Calculus [45]. They described a calculus with multiple interchangable semantics operating on a single biological syntax based on interaction diagrams. The syntax is formally represented by a collection of relations, sets, and an initial status. The various semantics describe by what means the relations are applied to the sets to generate a stream of data over time. The authors describe four semantics: continuous, differential equations, Michaelis-Menten, and stochastic. In the first three semantics, all possible relations that can be applied are applied at each step, with the difference being between what equations are

applied to update the data sets. In the stochastic semantics, only one relation (update) is applied at each step, chosen based on probabalistic factors.

Schilstra and Bolouri [57] describe a method for mapping boolean logic operations, such as AND and OR, into the domain of mathematical equations for biological modeling. The authors describe these models for both logical operations and for linear primitives which express the simpliest linear equation possible for certain operations.

Curit et al. devise an extension to π-calculus semantics that allows a formal representation of causality in concurrent systems [15]. They demonstrate this model showing a protein reaction where-in a series of reactions must occur in a particular order. The authors use the concept of causality to impose an ordering on reductions, ensuring that some reduction happens before another.

Danos and Laneve construct a calculus for the purpose of representing biological networks at the molecular level [16]. The calculus, called κ-calculus, consists of proteins, sites, and a function which maps proteins to a multiset of sites. This function describes the method of protein interaction. Proteins may also be composed to form complexes. The authors then define chemical reactions at the protein level using rewrite rules. Finally, the authors describe an algorithm for compiling κ-calculus into asynchronous π-calculus.

The authors later extend their work to include a graphical representation of κ-calculus [17]. They show proteins as being boxes, with the appropriate sites indicated on the boundary of the box. Depending on the interaction with other proteins, sites from one protein may be connected to another with a line. The rewrite rules are likewise replaced with visual rewrite rules that show a set of proteins and sites on the left and another set of the right. The authors then show

that the graphical κ-calculus can be represented with any graphical calculus which represents at least binary interactions and name creation.

Luca Cardelli introduces the Brane Calculus in [10]. The Brane Calculus is based on membranes, which form an encapsulation of computation. In this case, the membranes are highly fluid, and the operations of the calculus consist of modifications to the membranes. In addition, the calculus also supports the nesting of membranes. The precise set of actions differs depending on the application desired. For example, on the cellular level, the author gives phagocytosis (placing one membrane within another), pinocytosis, and exocytosis (expelling a membrane). On the molecular level, binding, release, and reaction of molecules are appropriate actions. Cardelli also shows how the Brane Calculus can represent the viral infection and hijacking of a cell.

5. CONCLUSION

The general-purpose languages COBOL, Fortran, and Lisp were originally conceived for domain-specific purposes (business logic, mathematical computation, and symbolic processing, respectively). Since then, domain-specific languages have enjoyed wide-spread use in a variety of fields. However, DSLs have several weaknesses, which become even more glaring as programming languages gain sophisticated type-checking and debugging features. Domain-Specific Embedded Languages (DSELs) offer an approach which combines the advantages of DSLs with the strengths of modern languages.

This thesis has presented a DSEL for probabilistic programming. We have shown how uncertainty is an important factor in many programs, and how modern programming languages offer little or no probabilistic abstraction. We have introduced a DSEL for probabilistic programming and have given examples in several domains. Our DSEL allows direct specification of uncertainty using probabilistic constructs, such as distributions, rather than requiring the programmer to explicitly represent their computation using random numbers.

Finally, we have presented an experience and application of the DSEL in use by the Center for Gene Research at Oregon State University, demonstrating that the probabilistic DSEL is not merely an academic exercise, but meets actual needs of users and is suitable for expressing complex, real-life models.

BIBLIOGRAPHY

[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] M. Antoniotti, F. Park, A. Policriti, N. Ugel, and B. Mishra. Foundations of a Query and Simulation System for the Modeling of Biochemical and Biological Processes. In *Pacific Symp. on Biocomputing*, pages 116–127, 2003.

[3] Jerry Banks, John S. II Carson, and Barry L. Nelson. *Discrete-Event Simulation*. Prentice Hall, 2nd edition, 1999.

[4] J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[5] E. Bertino, D. Bruschi, S. Franzoni, I. Nai-Fovino, and S. Valtolina. Threat Modelling for SQL Servers. In *8th IFIP Conf. on Communications and Multimedia Security*, pages 159–171, 2004.

[6] A. Bhan, D. J. Galas, and T. G. Dewey. A Duplication Growth Model of Gene Expression Networks. *Bioinformatics*, 18(11):1486–1493, 2002.

[7] Richard Bitter, Taqi Mohiuddin, and Matthew Nawrocki. *LabVIEW advanced programming techniques*. CRC Press, 2001.

[8] Bossel, Hartmut. *Modeling and Simulation*. A K Peters, 1994.

[9] Luca Cardelli. Bioware Languages. *Computer Systems: Theory, Technology, and Applications – A Tribute to Roger Needham, Monographs in Computer Science*, pages 56–65, 2004.

[10] Luca Cardelli. Brane Calculi. In *Computational Methods in Systems Biology*, 2004.

[11] Luca Cardelli. Languages for Systems Biology. In *Grand Chellenges UK, GC1 InVivo <=> InSilico*, 2004.

[12] J. C. Carrington and V. Ambros. Role of microRNAs in Plant and Animal Development. *Science*, 301:336–338, 2003.

[13] Bor-Yuh Evan Chang and Manu Sridharan. PML: Toward a High-Level Formal Language for Biological Systems. In *Bio-CONCUR*, 2003.

[14] Checkoway, Cheryl and Kirk, Kim and Sullivan, Donna and Townsend, Marianne, editor. *SIMULINK User's Guide*. The Math Works, 1993.

[15] Michele Curti, Pierpaolo Degano, and Cosima Tatiana Baldari. Causal π-Calculus for Biochemical Modelling. In C. Priami, editor, *Computational Methods in Systems Biology*, LNCS 2602, pages 21–33, 2003.

[16] Vincent Danos and Cosimo Laneve. Core Formal Molecular Biology. In *European Symp. On Programming*, LNCS 2618, pages 302–318, 2003.

[17] Vincent Danos and Cosimo Laneve. Graphs for Core Molecular Biology. In *Computational Methods in Systems Biology*, LNCS 2602, pages 34–46, 2003.

[18] Davis, Paul W. *Differential Equations: Modeling with MATLAB*. Prentice Hall, 1999.

[19] 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.

[20] M. Erwig and S. Kollmansberger. Probabilistic Functional Programming in Haskell. *Journal of Functional Programming*, 2005. To appear.

[21] M. Erwig and S. Kollmansberger. Modeling Genome Evolution with a DSEL for Probabilistic Programming. In *8th Int. Symp. on Practical Aspects of Declarative Languages*, 2006. To appear.

[22] Filinski, Andrzej. *Controlling Effects*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996.

[23] George S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer, 2001.

[24] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. IEEE Press, 2004.

[25] Jóse M. Garrido. *Object-Oriented Discrete-Event Simulation with Java*. Kluwer Academic / Plenum Publishers, 2001.

[26] Michael Andrew Gibson and Eric Mjolsness. Modeling the Activity of Single Genes. In James M. Bower and Hamid Bolouri, editors, *Computational Modeling of Genetic and Biochemical Networks*, chapter 1, pages 3–48. MIT Press, 2001.

[27] 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.

[28] Haefner, James W. *Modeling Biological Systems*. Int. Thomas Publishing, 1996.

[29] Hannon, Bruce and Ruth, Matthias. *Dynamic Modeling*. Springer, 2nd edition, 2001.

[30] William L. Harrison and Robert W. Harrison. Domain Specific Languages for Cellular Interactions. In *26th Annual IEEE Int. Conf. on Engineering in Medicine and Biology*, 2004.

[31] Eric C. R. Hehner. Probabilistic Predicative Programming. In *7th Int. Conf. on Mathematics of Program Construction*, volume 3125 of *LNCS*. Springer, 2004.

[32] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), 1996.

[33] Steven C. Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[34] Jones, Claire. *Probabilistic Non-determinism*. PhD thesis, University of Edinburgh, July 1990.

[35] Jones, Claire and Plotkin, Gordon D. A Probabilistic Powerdomain of Evaluations. In *4th IEEE Symp. on Logic in Computer Science*, pages 186–195, 1989.

[36] Na'aman Kam, David Harel, Hillel Kugler, Rami Marelly, Amir Pnueli, E. Jane Albert Hubbard, and Michael J. Stern. Formal Modeling of C. elegans Development: A Scenario-Based Approach. In C. Priami, editor, *Computational Methods in Systems Biology*, LNCS 2602, pages 4–20, 2003.

[37] Hiroaki Kitano. Systems Biology: Toward System-level Understanding of Biological Systems. In Hiroaki Kitano, editor, *Foundations of systems biology*, chapter 1, pages 3–36. MIT Press, 2001.

[38] Knuth, D. *TEX: The Program*. Addison-Wesley, 1986.

[39] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating Computer Simulation Systems*. Prentice Hall, 2000.

[40] 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.

[41] Harley H. McAdams and Lucy Shapiro. Circuit Simulation of Genetic Networks. *Science*, 269(5224):650–656, 1995.

[42] Robin Milner. *Communication and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[43] Mogensen, Torben. Roll: A Language for Specifying Die-Rolls. In *5th Int. Symp. on Practical Aspects of Declarative Languages*, pages 145–159, 2003. LNCS 2562.

[44] Morgan, Carroll and McIver, Annabelle and Seidel, Karen. Probabilistic Predicate Transformers. *ACM Trans. on Programming Languages and Systems*, 18(3):325–353, 1996.

[45] Masao Nagasaki, Shuichi Onami, Satoru Miyano, and Hiroaki Kitano. Bio-calculus: Its Concept and Molecular Interaction. In K. Asai, S. Miyano, and T. Takagi, editors, *10th Workshop on Genome Informatics*, volume 10, pages 133–143, 1999.

[46] 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.

[47] Park, Sungwoo and Pfenning, Frak and Thrun, Sebastian. A Probabilistic Language based upon Sampling Functions. In *31st Symp. on Principles of Programming Languages*, pages 171–182, January 2004.

[48] 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.

[49] Udo W. Pooch and James A. Wall. *Discrete Event Simulation*. CRC Press, 1993.

[50] Ramsey, Norman and Pfeffer, Avi. Stochastic Lambda Calculus and Monads of Probability Distributions. In *29nd Symp. on Principles of Programming Languages*, pages 154–165, January 2002.

[51] Amitai Regev and Ehud Shapiro. Cells as Computation. In C. Priami, editor, *Computational Methods in Systems Biology*, LNCS 2602, pages 1–3, 2003.

[52] 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.

[53] Renshaw, Eric. *Modelling Biological Populations in Space and Time*. Cambridge University Press, 1993.

[54] Stewart Robinson. *Simulation: The Practice of Model Development and Use*. John Wiley and Sons, 2004.

[55] Rubinstein, R. Y. *Simulation and the Monte Carlo Method*. Wiley, 1981.

[56] T. Sato and Y. Kameya. Parameter Learning of Logic Programs for Symbolic-Statistical Modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

[57] Maria J. Schilstra and Hamid Bolouri. The Logic of Gene Regulation. In *3rd Int. Conf. on Systems Biology*, 2002.

[58] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S. Baliga, Jonathan T. Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Research*, 13(11):2498–2504, 2003.

[59] Sigmon, Kermit. *MATLAB Primer*. CRC Press, 5th edition, 1998.

[60] Ricard V. Sole, Romualdo Pastor-Satorras, Eric Smith, and Thomas B. Kepler. A Model of Large-scale Proteome Evolution. *Advances in Complex Systems*, 5:43–54, 2002.

[61] Starfield, Anthony M. and Smith, Karl A. and Bleloch, Andrew L. *How to Model It: Problem Solving for the Computer Age*. Burgess Publishing, 2nd edition, 1994.

[62] C. Talcott, S. Eker, M. Knapp, P. Lincoln, and K. Laderoute. Pathway Logic Modeling of Protein Functional Domains in Signal Transduction. In *Pacific Symp. on Biocomputing*, pages 568–580, 2004.

[63] Sarah A. Teichmann and M. Madan Babu. Gene Regulatory Network Growth by Duplication. *Nature Genetics*, 36(5):492–496, 2004.

[64] S. A. Thibault, R. Marlet, and C. Consel. Domain-specific languages: From design to implementation application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3), 1999.

[65] Jeffrey Travis. *LabVIEW for Everyone*. Prentice Hall, 2nd edition, 2002.

[66] A. Vazquez, A. Flammini, A. Maritan, and A. Vespignani. Modeling of Protein Interaction Networks. *ComPlexUs*, 21:38–44, 2003.

[67] Wilson, Will. *Simulating Ecological and Evolutionary Systems in C*. Cambridge University Press, 2000.

[68] Eric P. Xing, Wei Wu, Michael I. Jordan, and Richard M. Karp. LOGOS: a Modular Bayesian Model for de novo Motif Detection. In *IEEE Computer Society Conf. on Bioinformatics*, pages 266–276. IEEE Computer Society, 2003.

APPENDICES

APPENDIX A. Overview of Monads

Functional programming abstracts away two attributes common to other languages: sequencing and side effects. Sequencing refers to the important of evaluating certain applications in order, as one may affect the outcome of another. Normally, functional programs are evaluated lazily, with evaluation of any given term delayed as long as possible. Functional programs also have, generally, no side effects. This means that the evaluation of one term cannot have any effect on the result of evaluating some other term. Sometimes, however, it is important for one computation to affect the next. Monads provide a controlled, specific method for sequencing computation and organizing side effects.

A.1. The Maybe Monad

To see an example of monads in action, consider first the datatype `Maybe` which is used to indicate a computation that might fail.

```
data Maybe a = Just a | Nothing
```

If the computation succeeds, the `Just` value is returned, otherwise `Nothing` is returned. Consider the following function which divides two integers.

```
divide :: Int -> Int -> Int
divide x y = x `div` y
```

However, if `y` was zero, a run-time error would result. We could avoid this by making `divide` return a `Maybe` type, which would indicate `Nothing` if the divisor was zero.

```
divide :: Int -> Int -> Maybe Int
divide x y = if y == 0 then Nothing else Just (x `div` y)
```

Now imagine we want a function which takes three numbers: x, y and z. It divides x by z and y by z then adds the results together. We might first try:

```
divxyz :: Int -> Int -> Int -> Int
divxyz x y z = (divide x z) + (divide y z)
```

However, this does not work because `divide` returns `Maybe Int`, which is not a numeric type. Instead, we need to use case statements to extract the (possible) value, or handle the failure.

```
divxyz :: Int -> Int -> Int -> Maybe Int
divxyz x y z = case (divide x z) of
        (Just xz) -> case (divide y z) of
                (Just yz) -> Just (xz + yz)
                Nothing -> Nothing
        Nothing -> Nothing
```

What a mess! If we look at this a bit, we can see a pattern. Whenever a computation returns `Just`, extract the value and continue. If the computation returns `Nothing`, abort and do not perform any further computation. In this case, the outcome of one computation (say, `divide x z`) affects another (`divide y z`). In particular, the result of the first computation could cause the second to possibly not be executed.

Wouldn't it be nice if there were a way to take this behavior and abstract it away from the particular application, so that we wouldn't have to have tons of `Just`s and `Nothing`s all over the place? How would this work? We can start by defining a way to combine `Maybe` computations.

```
bind :: Maybe a -> (a -> Maybe b) -> Maybe b
bind Nothing _ = Nothing
bind (Just x) f = f x
```

The function `bind` takes a `Maybe` value and a function which operates on the value inside, producing another `Maybe` value. If the first value is `Nothing`, there's

no need to evaluate the function. Otherwise, extract the value and apply the function. We could use this to rewrite the above code.

```
divxyz :: Int -> Int -> Int -> Maybe Int
divxyz x y z =
  bind (divide x z) (\xz->
        bind (divide y z) (\yz->
        Just (xz + yz) ))
```

This is already a substantial improvement. We have abstracted away the handling of `Nothing` and are left with just the logic. In general, this form of computation is known as a *monad*.

In Haskell, monads are defined as a type class. For a data type to be an instance of `Monad`, two functions must be defined: `return`, which injects a value into the monad, and `>>=` (pronounced bind), which connects monadic computations.

```
class Monad m where
        return :: a -> m a
        (>>=) :: m a -> (a -> m b) -> m b
```

We can make `Maybe` an instance of the type class `Monad` using the definition of `bind` we already derived. What would be the definition of `return`? Simply applying the `Just` constructor.

```
instance Monad Maybe where
        return x       = Just x
        Nothing  >>= _ = Nothing
        (Just x) >>= f = f x
```

Haskell supports a special shorthand notation for monads called the do-notation. Using the do-notation, we can rewrite `divxyz` into a nice, sequenced function.

```
divxyz :: Int -> Int -> Int -> Maybe Int
divxyz x y z = do
        xz <- divide x z
        yz <- divide y z
        return (xz + yz)
```

This is much clearer than our tedious original attempt with case statements. In particular, the handling of all the `Nothing` cases is completely abstracted away and hidden from the user of the monad.

A.2. The List Monad

A list is simply a sequence of values. In some cases, we might want to apply a function to each value in a list. This can be done with the function `map`. For example, if we wanted to increase a list of numbers each by 1, we could define a function (`incr1` below) which maps the successor function to a list.

```
incr1 xs = map succ xs
```

```
> incr1 [1,2,3]
[2,3,4]
```

Imagine now we have two lists, and we want to combine all possible pairs, for example, to find all the sums of one element from the first list and one from the second.

```
ls xs ys = concatMap (\x->map (\y->x+y) ys) xs
```

Like the `Maybe` example, there is a lot of overhead here—we're creating a function and then applying to each element using a `concatMap`, which maps a function to a list, then concatenates all resulting lists into a single list. The innermost value only uses a `map` because it produces only one value. The logic here (`x+y`) is shrouded in organizational overhead.

Like with the `Maybe` monad, we can hide away the overhead inside a monad. How can we combine lists? We want to apply the function `f` to each element in a given list, and concatenate all the results together to produce a single list of results.

```
bind :: [a] -> (a -> [b]) -> [b]
bind la f = concatMap f la
```

This is precisely how the monad is defined.

```
instance Monad [] where
        return x = [x]
        m >>= f = concatMap f m
```

Using the previously shown do-notation, we can rewrite our function in a much more attractive and comprehensible form.

```
ls xs ys = do
        x <- xs
        y <- ys
        return (x+y)
```

The monad presented in this thesis can be considered as an extension to the list monad.

APPENDIX B. Source Code Availability

The complete code for the probabilistic programming DSEL along with examples is available for download at

`http://eecs.oregonstate.edu/~erwig/pfp/`

Source code for the genome evolution application (which requires the PFP DSEL) is available by e-mail, `kollmast@eecs.orst.edu`