
All of Probability in a Box

0

Chapter

Contents

1	Fixed Random Payoffs	1
1.1	Kinds	3
1.2	Overview of frplib	7
1.3	Predictions and Prices	8
1.4	Kinds as Models	18
1.5	The Big 3+1!	31
2	Transforming with Statistics	35
2.1	Statistics and Data Processing	36
2.2	Transformed Kinds	51
2.3	Projections and Marginals	57
2.4	Examples	63
2.5	frplib Statistics: Builtins, Factories, and Combinators	96
3	Equivalent Kinds and Canonical Forms	108
4	Building with Mixtures	122
4.1	Independent Mixtures	126
4.2	Conditional FRPs and Conditional Kinds	148
4.3	General Mixtures	158
5	Constraining with Conditionals	197
6	Three Dialogues: Computation, Systems, Simulation	228
6.1	A Dialogue on Computation	228
6.2	A Dialogue on Systems and State	239

6.3	A Dialogue on Solutions and Simulation	254
7	Predicting with Expectations	273
7.1	Fundamental Properties of Risk-Neutral Prices	283
7.2	Computing Expectations	307
7.3	Kinds and Expectations	316
7.4	Probabilities are the Expectations of Events	321
8	Patterns, Predictions, and Practice	324
8.1	Types and Operations	325
8.2	Simple Finite Random Processes	327
8.3	Strategies and Representations	346
8.4	Using Observations	355
8.5	Touching Infinity	363
	Index	376

1 Fixed Random Payoffs

Key Take Aways

A **Fixed Random Payoff** box, or **FRP** for short, is a device that does one thing, one time: it produces a value. Once the value is produced it is fixed for all time, but before that it is uncertain, non-deterministic, ... *random*. The value represents the promise of a payoff to the FRP's owner, and the question of how much an FRP is *worth* hinges on how well we can predict its value.

Since an FRP produces a single value by mysterious means, we need more information to predict that value effectively. Fortunately, each FRP has a **Kind**, and we have access to an unlimited supply of FRPs of each Kind. By studying *in aggregate* the values of many FRPs of the same Kind, we can build a deeper understanding of how to predict an individual FRP's value.

An FRP Kind is a complete tree with a positive, numeric **weight** on each edge and a distinct **value** at each node. The leaf nodes give the possible values that the FRP can produce; each value is a list, usually a list of one or more numbers. A path from the root to the leaf shows a sequence of items being successively added to the list, which starts out empty at the root. The values at every leaf node must be lists of the same length; this length is called the **dimension** of the FRP and its Kind. When the dimension is 1, we have a **scalar** FRP and Kind. The number of leaf nodes (and thus possible values) is called the **size** of the FRP and its Kind.

An FRP is a closed box whose top face has a single button, an LED, several ports, a touch-screen display, and a smaller metallic display. (Figure 1.) We can neither open the box nor see what is inside it, directly or indirectly.

An FRP does one thing, one time – it produces a value. Before the button has ever been pushed, the FRP is *fresh*, with both the Observed Indicator LED and the Display off. The first time the button is pushed, the LED turns on and remains *steadily on thereafter*. The Display then turns on for a few moments and shows a value. Whenever the button is pushed thereafter, the display turns on for a few moments and shows *that same value*.

Before you push the button, you do not know what that value will be, and once

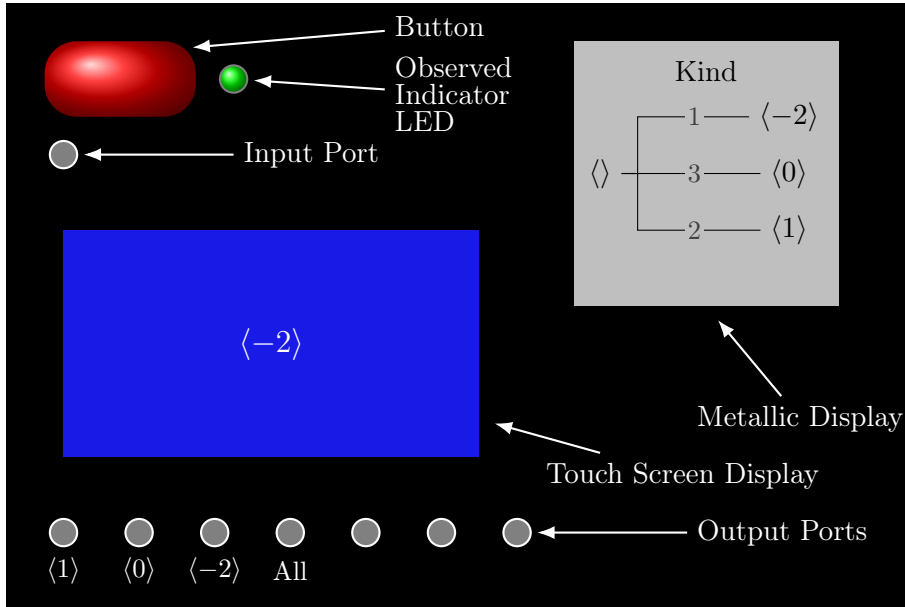


FIGURE 1. The top face of a typical fixed random payoff box, or FRP. The text will explain the role of each element shown here, including Kinds in section 1.1 and the input/output ports in Section 4 and 2.

you’ve pushed the button, the value is *fixed* for all time (the F in FRP). You do not know where the value comes from or how it was produced. It could be the result of a complex physical process or a value that a group of gnomes living inside the box finds amusing. The FRP’s output value is *random* in a sense we will explore (the R in FRP). If you own an FRP, you hold the *promise* of receiving its value as a *payoff* (the P in FRP).

It is possible to have FRPs whose values are lists of any type, but we will almost always use *numeric FRPs* that output *lists of numbers*. We require that all possible values that might be output by any particular FRP are lists of the *same length*. Throughout, we will use angle brackets $\langle \rangle$ to denote lists and tuples.¹ For example, $\langle \rangle$ is the empty list, $\langle 1 \rangle$ has a single element 1, $\langle 0, 1 \rangle$ has two elements with first element 0, $\langle -1, 2, 32 \rangle$ has three elements with first element -1, and so forth.² The *length* of a list is the number of elements it has. A list of length 1 is called a **scalar**, and in practice, we treat these specially by making no distinction between a list with one element and the quantity it contains. If I give you $\langle 42 \rangle$, then for all practical purposes I have given you the number 42, and vice versa. We say that an FRP has **dimension**

¹It is common to use parentheses for lists, tuples, and vectors like $(3, 4, 5)$. This is fine, but parentheses are so frequently used and overloaded that it is helpful to have a more salient delimiter for this purpose.

²See F.7 and F.9.2 for more on lists, tuples, and vectors.

n if its outputs are lists of length n . If it has dimension 1, we call it a **scalar FRP**.

How do we interpret an FRP's value (fixed for all time yet randomly produced)? As the P in FRP indicates, an FRP is a *promise* of a *payoff*. If you own a scalar FRP with value v , you are entitled to receive $\$v$ one time. If $v < 0$, this entails an obligation to pay $\$|v|$. (For an FRP of dimension $n > 1$, we think of a value $\langle v_1, v_2, \dots, v_n \rangle$ as a suite of n payoffs. But more on that later; for the moment, concentrate on the scalar case.)

How much is an FRP worth? What would you pay to own the promised payoff *before* seeing its value? Your answer is a *prediction* about the FRP's value. For instance, you would not be willing to pay a high price if you were quite certain that you would lose money on the deal; nor would you reject an offered price if you were quite certain to make money. A good prediction incorporates all the information we have about the FRP at a given time, but the accuracy of any prediction increases with our uncertainty in the output value. Prediction. Accuracy. Uncertainty. These are all words that we will need to define more precisely. Understanding how much an FRP is worth – that is, how to find our best prediction of its value – is at the core of probability theory and will be a central goal for the rest of this Chapter.

Because each FRP produces just a single value in mysterious ways, we need more information to make good predictions about its value. This is where the *Kind* comes in, as depicted on the FRP's metallic display. We will see that FRPs with the same Kind behave similarly in some sense, so by considering a large collection of FRPs of the same Kind, we can learn what we need to make good predictions. Indeed, it turns out that the Kind itself gives enough information to make any sort of prediction about any FRP with that Kind.

The ability to assess an FRP's worth – and thus make informed predictions about its value – has manifold applications in the face of uncertainty. We can design FRPs to describe and model a huge variety of random systems and processes, and our predictions inform decisions and actions that can guide us toward desirable outcomes.

1.1 Kinds

An FRP's **Kind** embodies our knowledge about the FRP's value. Two fresh FRPs *of the same Kind* are, all else being equal, interchangeable. describes the nature of the FRP in a way that will let us make good predictions about the FRP. A Kind

is represented by a *rooted tree*³ where each edge has an associated positive number. The root node holds the empty list $\langle \rangle$. The leaf nodes hold all the possible values that an FRP with that Kind might output. Within each level of the tree, the nodes must hold *distinct* values that are lists of the same type and length. A tree representing a Kind must also be *complete*, meaning that every path from the root to a leaf has the same number of steps. Along each such path, successive nodes differs from their predecessors by appending a single item. The positive numbers associated with the tree's edges are called the Kind's **weights**.

³A connected, acyclic, undirected graph with one node designated as the “root.” See Interlude G.

Figure 2 shows a few simple examples of Kinds. We display Kinds horizontally with the root at the *left* and the leaves at the *right* rather than the more common root-at-the-top display. This layout offers several advantages for us, including compactness, easier comparison of values and weights, and simpler output for the programs we use. (Figures 2 and 3 show the same Kinds with horizontal and vertical layout to help you get used to the horizontal layout.) FRPs of the left Kind in Figure 2 can only ever output the single value $\langle 1 \rangle$. Those of the middle Kind in Figure 2 can output either $\langle -1 \rangle$ or $\langle 1 \rangle$, and those of the right Kind can output $\langle -1 \rangle$, $\langle 0 \rangle$, or $\langle 9 \rangle$. Again, we do not distinguish in practice between tuples with one number and the number itself, so we can think of FRPs with these Kinds as outputting random numbers from the sets $\{1\}$, $\{-1, 1\}$, and $\{-1, 0, 9\}$.

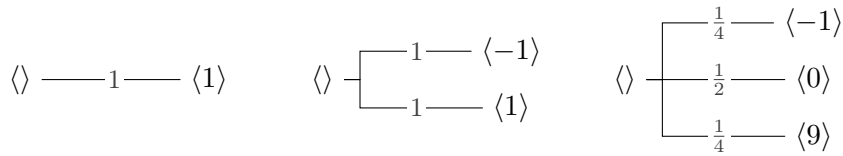


FIGURE 2. Several FRP Kinds, all with dimension 1 (scalar) and with sizes 1, 2, and 3.

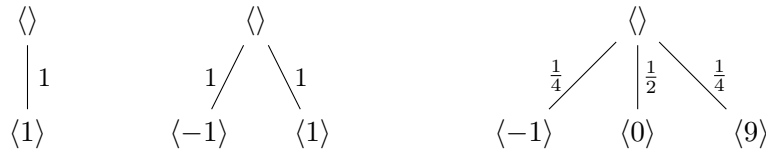


FIGURE 3. The same Kinds as in Figure 2 but displayed with the root at the top.

Figure 4 shows a more complicated example. FRPs of this Kind can output the possible values $\langle -1, -15 \rangle$, $\langle -1, -5 \rangle$, $\langle 0, 10 \rangle$, $\langle 9, 12 \rangle$, $\langle 9, 20 \rangle$, or $\langle 9, 32 \rangle$. We think of the

values on the leaves as being generated in stages. Starting from an empty list at the root, the FRP first generates one of -1, 0, or 9 and appends it to the list, producing one of the lists shown at the first level. Then depending on whether -1, 0, or 9 was appended, generates another number (-5 or -15, 10, and 12 or 20 or 32 respectively) and appends it to the list. The list at each node contains the numbers generated in order along the path from the root. Remember: *at each level, the generated tuples must be distinct and the weights positive.*

FRPs and Kinds have several properties that we refer to frequently.

Property	FRP	Kind
dimension d	all values have length d	every leaf has length d
size s	s distinct, possible output values	s leaf nodes
type $0 \rightarrow d$	has dimension d	has dimension d
width w	—	w edges along any path from root to leaf

(The 0 in the type will be explained later.) An FRP and its Kind always have the same size, dimension, and type.

The Kinds in Figure 2 have, respectively: size 1, 2, and 3. They all have width 1 and dimension 1. The Kind in Figure 4 has size 6, width 2, and dimension 2. The trivial FRP, called **empty**, has size 1 and dimension 0; its Kind, denoted by $\langle \rangle$ is just a root node with an empty list, with size 1, width 0, and dimension 0. We call FRPs (and Kinds) with dimension 1 *scalar* FRPs (and Kinds).

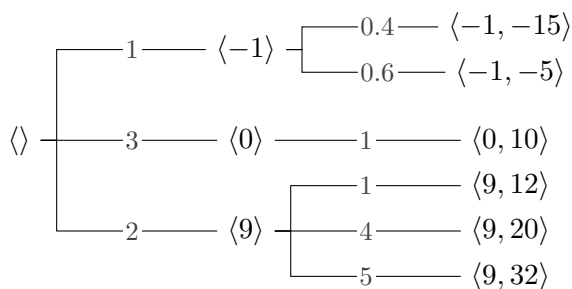


FIGURE 4. A Kind with more than two levels and possible values of dimension 2.

For a Kind like that in Figure 4, with width is bigger than 1, the structure of the tree gives us a picture of a random process at work “in stages”. We can think of the FRP as generating its value sequentially, starting from the empty tuple at the

root and appending a new number to the list at each level along a path from root to leaves, where the number generated at each stage is contingent on the previous numbers in the list. So, the tuple at each node shows the numbers generated so far on the path to that node, with the last element having been generated and appended at that level, eventually yielding a final value at a leaf node.

Because a Kind's dimension can be bigger than its width, we can also think of an FRP as generating its value “all at once.” Figure 5 shows a dimension 2 Kind with width 1 and the same possible values as the Kind of Figure 4. In this view, the FRP simply spits the whole tuple when activated.

It turns out that these two perspectives on Kinds – describing values produced “in stages” or “all at once” – are consistent. We can view the Kind of a multi-dimensional FRP in either way. Section 3 will show that the Kinds in Figures 4 and 5 are in fact *equivalent*: they give the same predictions and we can convert seamlessly between them.

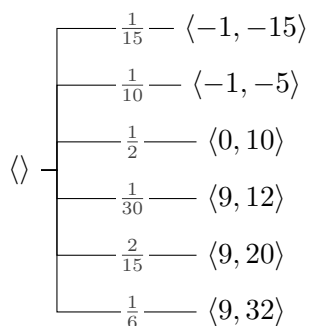


FIGURE 5. A Kind with dimension 2 and width 1.

Puzzle 1. Draw the Kind of an FRP of dimension 2 in which knowing the second component of the value gives you no information about the first component.

Puzzle 2. Draw the Kind of an FRP of dimension 3 (and size bigger than 3) in which the values generated are all lists whose elements sum to zero.

We usually give FRPs names that start with a capital letter, often using only a single capital letter, possibly adorned with subscripts or labels.

1.2 Overview of frplib

Talking about FRPs and Kinds is useful, but we get even more by building FRPs and Kinds, manipulating them, and using them to compute predictions for interesting random systems. For this purpose, we will use software: **frplib** is a package built on the Python programming language that can be used both as an *interactive laboratory* for working with FRPs and Kinds or as a *library* within a standalone program. This system encourages you to play with the examples, check your understanding against output, and engage more deeply with the ideas.

Instructions for downloading and installing **frplib** are available at <https://github.com/genovese/frplib>.⁴ When the package has been installed, two things happen: from within your Python programs, you will be able to import functions and data from **frplib**, and the application **frp** will be installed on your system. The **frp** application is run from your terminal command line and opens an interactive session for computing with FRPs, Kinds, and related quantities.

⁴Mac OS, Windows, and Linux are all supported.

Within a Python program, you can load modules from the **frplib** library. For example:

```
from frplib.frps      import FRP, frp, conditional_frp
from frplib.kinds      import Kind, kind, constant, either, uniform
from frplib.statistics import statistic, __, Proj, Sum
```

The library documentation describes the available modules, functions, and data, along with shortcuts for importing some commonly used configurations. Modules within the sub-package **frplib.examples** include functions and data used in examples within this book. You will see such modules loaded or mentioned in the text and are encouraged to follow along in **frplib** as you read.

The terminal application **frp** is invoked with various *sub-commands* that open different interactive environments. The two we will focus on are the **market** and **playground** sub-commands. To invoke these sub-commands, enter **frp market** and **frp playground** at the terminal command line prompt. (There are several ways to start the application, as described in the instructions on the GitHub page, including **frp** and **python3 -m frplib**, but here, we will use the former as a placeholder.)

When you enter **frp market**, you will see a prompt **market>** at which you can enter tasks for the market to perform. These can span multiple lines and must end

in a period (.). After the first line of a multi-line task, the prompt will change to `...>` signaling that you can continue entering information. A `.` at the end of a line will complete the input. However, if the input is ill-formed in any way, the task will not be submitted; instead, an error message will identify the problem, allowing you to fix it. To end your session, enter `“exit.”` or `“done.”` at a fresh prompt. Enter `“help.”` for assistance.

When you enter `frp playground`, you will see a prompt `playground>` at which you can enter commands or other Python code. This code can span multiple lines; you end a code block by hitting return on a blank line. In the playground, you can move across the code, even multiple lines, and edit it before final submission. You can also access and edit your interaction history. The playground pre-loads all the most commonly used `frplib` functions and classes, so you can use them easily. In addition, you can import any `frplib` or other installed Python package from the playground prompt. You can use the built-in `help` system on any defined function or object; in addition, the function `info()` gives `frplib`-specific help. Start with `info("overview")`.

The `frplib` library provides classes and methods relating to all the main concepts we cover in this chapter, including FRPs, Kinds, and Statistics. For each of these, the library defines **factories**, which are functions for creating objects of the specified type with particular properties; **combinators** for combining several objects of a specified type into a new one; along with various **actions** and **utilities**. See the “Playground Overview” on page 106 and the `frplib` Cheatsheet for a summary, along with the many examples in this chapter.

1.3 Predictions and Prices

You have access to the FRP Warehouse, an organization that can provide a seemingly inexhaustible supply of FRPs. The Warehouse manager does not like other people poking around, so you tell the manager the *Kind* and number of FRPs you want, and the manager fabricates them for you. All these FRPs are *fresh*; their buttons have not ever been pushed.

It is rather tiresome to go to the Warehouse and haul back tons of boxes whenever you order some FRPs, not to mention pushing all the buttons and recording the values. Fortunately, the Warehouse is a highly automated operation, so you can manage

the entire transaction with software that the Warehouse makes available. With the `frp market` command, you can request any number of fresh FRPs, have their buttons pushed, and receive a record of the values from each (as a list or summary). Fast and painless for you, though a lot of work for the Warehouse staff.⁵

The Warehouse has given you a free trial, allowing you to get as many FRPs as you like at no cost and see their values. As this is only a trial, you receive no actual payoff in exchange for the FRPs you activate, but the free trial can help you understand how FRPs work, what their Kinds mean, and how to price them. Later, when the trial expires, money will change hands, so the stakes will be higher.

Some of the market tasks operate on a Kind and need you to specify the Kind of FRP to simulate. For this, the market program uses a simple text format⁶ that represents the tree. For example, the Kinds in Figures 2 and 4 are represented by the following strings:

```
(<> 1 <1>)

(<> 1 <-1> 1 <1>)

(<> 0.25 <-1> 0.5 <0> 0.25 <9>)

(<> 1 (<-1> 0.4 <-1, -15> 0.6 <-1, -5>)
  3 (<0> 1 <0, 10>)
  2 (<9> 1 <9, 12> 4 <9, 20> 5 <9, 32>))
```

Each string is a $()$ -balanced expression with weights and values in alternating pairs at each level and each value tuple is enclosed in `<>`. Subtrees are enclosed in parentheses and start with the subtree’s root. The weight precedes its associated value or subtree in the list. Whitespace, including any newlines, is ignored.

When displaying input to the market, we will always abbreviate the “`market>`” prompt as “`mkt>`” to save space. Prompts like `...>` are continuations of input over multiple lines. Output follows the task that generates it.

We use the `show` task to check that our input string gives the Kind we expect.

```
mkt> show kind (<> 1 <-1> 1 <1>).
,----- 1 ----- <-1>
<> -|
`----- 1 ----- <1>
```

⁵Many Bothans worked hard to bring you this information.

⁶Details can be seen with the “`help kinds.`” task in the market.

```
mkt> show kind (<> 1 (<0> 1 <0, 0> 2 <0, 1> 3 <0, 2>)
...>                2 (<1> 1 <1,1> 1 <1, -1>)).
```

```

,----- 1 ----- <0, 0>
,----- 1 ----- <0> +----- 2 ----- <0, 1>
|                        `----- 3 ----- <0, 2>
<> +
|                        ,----- 1 ----- <1, -1>
`----- 2 ----- <1> |
                        `----- 1 ----- <1, 1>
```

The market will detect ill-formed syntax and give you a chance to fix it:

```
mkt> show kind (<> 1)
The input '(<> 1)' is not a valid kind; it appears to be missing a value.

mkt> show kind (<> 1 <1> 2 <1>)
The input '(<> 1 <1> 2 <1>)' is not a valid kind; its values are not unique.

mkt> show kind (<> 1 <1> 2 <1, 2, 3>)
The input '(<> 1 <1> 2 <1, 2, 3>)' is not a valid kind; its values do not
have the same dimension.
```

These error messages might show up in a highlight color at the bottom of the window.

Now let's begin by examining the simplest, non-empty FRP, which always outputs the same value. This has Kind $\langle \rangle$ —1— $\langle 1 \rangle$. To examine 10,000 FRPs with this Kind, we enter a task in the market:

```
mkt> demo 10000 with kind (<> 1 <1>).
Activated 10000 FRPs with kind
<> -+----- 1 ----- <1>
Summary of Output Values
|-----+-----+-----|
| Values   | Count | Proportion |
|-----+-----+-----|
```

```
| <1>          | 10000 | 100%          |
|-----+-----+-----|
```

This tells us that all 10000 of the FRPs of this Kind gave value 1, which makes sense as that is the only possible value they can give. Try this again with a different number of FRPs in the demo.

Next, let's vary the weight. For instance, try:

```
mkt> demo 10000 with kind (<> 0.001 <1>).
```

What do you get? Try it with a variety of different weights. Formulate a hypothesis to answer the following question.

Puzzle 3. For a Kind of size 1, what can you say about the output of a demo of FRPs with that Kind? How do the weights influence the results?

The Kind $\langle \rangle \text{ --- } w \text{ --- } \langle v \rangle$ is named **constant**(v). An FRP with that Kind is called a **constant FRP** with value v because it always outputs the value v . For all weights w , it can be completely identified with the constant v itself; whether I gave you the value v or an FRP that produces that value, you should be indifferent. For all practical purposes, they are the same. Indeed, in this special case, we can abuse our notation a bit and display the Kind **constant**(v) without the (irrelevant) weight, which we will implicitly take to be 1: $\langle \rangle \text{ --- } \langle v \rangle$

Let us look at Kinds with more structure, restricting our attention for the moment to Kinds of width 1. The next simplest case is size 2. Consider the Kind

$$\langle \rangle \begin{cases} \text{---} 1 \text{ --- } \langle 0 \rangle \\ \text{---} 1 \text{ --- } \langle 1 \rangle \end{cases}$$

and run a demo where you examine the values of 10,000 FRPs of this Kind. Here's the command and an output similar to what you will see:

```
mkt> demo 10000 with kind (<> 1 <0> 1 <1>).
Activated 10000 FRPs with kind
      ,----- 1 ----- <0>
<> -|
      `----- 1 ----- <1>
```

Summary of Output Values

Values	Count	Proportion
<0>	5031	50.31%
<1>	4969	49.69%

The numbers of 0's and 1's are almost equal. Trying it with a larger number of FRPs, say one million,⁷ yields

```
mkt> demo 1_000_000 with kind (<> 1 <0> 1 <1>).
```

Activated 1000000 FRPs with kind

```
,----- 1 ----- <0>
<> -|
      `----- 1 ----- <1>
```

Summary of Output Values

Values	Count	Proportion
<0>	499895	49.99%
<1>	500105	50.01%

The counts in these tables will vary slightly with each demo because they are based on a sample of FRPs, each with its particular value, and the specific contents of a sample determine the proportions we see. As a demo gets larger, this “sampling” variation gets smaller.

Now vary the weights over a wide range of possibilities, for instance:

```
mkt> demo 1_000_000 with kind (<> 2 <0> 2 <1>).
mkt> demo 1_000_000 with kind (<> 100 <0> 100 <1>).
mkt> demo 1_000_000 with kind (<> 0.5 <0> 0.5 <1>).
mkt> demo 1_000_000 with kind (<> 0.01 <0> 0.01 <1>).
mkt> demo 1_000_000 with kind (<> 1 <0> 4 <1>).
mkt> demo 1_000_000 with kind (<> 1 <0> 9 <1>).
```

⁷In the market, numbers can contain _ to separate blocks of three digits and make the numbers more readable.

```

mkt> demo 1_000_000 with kind (<> 19 <0> 1 <1>).
mkt> demo 1_000_000 with kind (<> 2 <0> 8 <1>).
mkt> demo 1_000_000 with kind (<> 1900 <0> 100 <1>).
mkt> demo 1_000_000 with kind (<> 400 <0> 100 <1>).
mkt> demo 1_000_000 with kind (<> 0.38 <0> 0.02 <1>).
mkt> demo 1_000_000 with kind (<> 3 <0> 27 <1>).
mkt> demo 1_000_000 with kind (<> 0.2 <0> 0.8 <1>).
mkt> demo 1_000_000 with kind (<> 0.01 <0> 0.04 <1>).

```

and so on. You need not use 1,000,000 if you want quicker response, but keep the number of FRPs large like 100,000.

Puzzle 4. Based on the results of these explorations, formulate a hypothesis about the relationship between the weights in a size 2 Kind and the proportions you see in the demo of FRPs with that Kind.

As you try to formulate a hypothesis here, a geometric approach may help. View each set of weights you demo as a point $\langle w_0, w_1 \rangle$ in the plane. Running the demo gives a pair of proportions $\langle 1 - p_1, p_1 \rangle$ that sum to 1, where p_1 is the proportion of 1s in the table. Try plotting each point $\langle w_0, w_1 \rangle$ with a color indexed by the p_1 from the demo. When you run many demos with varied weights, what does this plot look like?⁸ What does it suggest about the relationship between the weights and the proportions in a demo with a very large number of FRPs?

⁸Example F.5.9 and the discussion around equation (F.5.15) in Interlude F are relevant here.

For the following two puzzles, use the market to explore the relationship using what you have learned in the size-2 case.

Puzzle 5. If I give you a choice between two FRPs, with actual payoff, with Kinds $\langle \rangle 10 \langle -1 \rangle 30 \langle 0 \rangle 20 \langle 10 \rangle$ and $\langle \rangle 100 \langle -1 \rangle 300 \langle 0 \rangle 200 \langle 10 \rangle$, which do you prefer and why? Back up your preferences with evidence from the `frp market`.

Puzzle 6. If you demo a large number of FRPs with Kind $\langle \rangle a \langle 0 \rangle b \langle 1 \rangle c \langle 2 \rangle$, where a , b , and c are arbitrary positive weights, what relative frequencies of the values 0, 1, and 2 do you expect to see?

Try it for various values of a , b , and c . Did your intuition match the results? What happens if you increase or decrease the number of FRPs you sampled?

Now, formulate a general hypothesis about what the weights mean. What is the relationship between the weights that you specify in the demo and the (ideal) proportions you see in the table? Use the market to test your hypothesis with a variety of Kinds of dimension and width 1, including those with size bigger than 3.

Puzzle 7. After formulating, possibly revising, and confirming your hypothesis about the meaning of the weights in a (width 1) Kind, write down your conclusion in sentence or two.

If we run a demo of n FRPs of size s whose Kind has weights w_1, w_2, \dots, w_s and values v_1, v_2, \dots, v_s , we will get a table of associated proportions p_1, p_2, \dots, p_s that sum to 1. Across repeated runs of this demo, these proportions will vary slightly because different FRPs are being activated, even if though the FRPs have the same Kind. As we make n larger, this “sampling variability” gets ever smaller, and the proportions p_i get closer to “ideal” proportions \bar{p}_i that are determined by the Kind’s weights. In fact,

$$\bar{p}_i = \frac{w_i}{w_1 + w_2 + \dots + w_s}, \quad (1.1)$$

or put another way, $\bar{p}_i/\bar{p}_j = w_i/w_j$. The proportions we see in the table are thus determined by *the weights normalized to sum to 1*, and the relative proportions of any two values are determined by the *relative magnitudes of the corresponding weights*.

The market’s buy task can be useful for leveraging this insight to find what we will call the “risk-neutral price” of an FRP. It allows one to purchase FRPs of specified Kinds and numbers at specified prices. It is like demo except it also computes the *net payoff* – the difference between the total payoff from all the purchased FRPs and the total payment we made for those FRPs – and the net payoff per FRP purchased for the entire demo. For example:

```
mkt> buy 1_000_000 @ 1 with kind (<> 2 <-5> 3 <0> 5 <4>).
```

Buying 1,000,000 FRPs with kind (<> 2 <-5> 3 <0> 5 <4>) at each price

Price/Unit (\$)	Net Payoff (\$)	Net Payoff/Unit (\$)
1.00	-1,319.00	-0.001319

Suppose we run a **buy** task with price c for each of n FRPs of size s whose Kind has weights w_1, w_2, \dots, w_s and values v_1, v_2, \dots, v_s , and the underlying demo has proportions p_1, p_2, \dots, p_s for those values. Then our net payoff per unit is

$$\frac{np_1v_1 + np_2v_2 + \dots + np_sv_s - nc}{n} = p_1v_1 + p_2v_2 + \dots + p_sv_s - c.$$

For large n , the p_i 's will be close to the “ideal” proportions \bar{p}_i derived from the weights. When we choose $c = \sum_{i=1}^s \bar{p}_i v_i$ as the price per FRP, the net payoff will be close to 0. This is the *risk-neutral price* of FRPs with that Kind, as we will discuss in detail in Section 7.

So far in this subsection, we have restricted our attention to Kinds of width and dimension 1 because these are simpler. Changing the dimension does not really affect our conclusions as the proportions are associated with values, whatever they are.

```
mkt> demo 1_000_000 with kind (<> 1 <1, 2, 3> 1 <9,11,16>).
```

```
Activated 1000000 FRPs with kind
```

```
,----- 1 ---- <1, 2, 3>
<> -|
      `----- 1 ---- <9, 11, 16>
```

```
Summary of Output Values
```

Values	Count	Proportion
<1, 2, 3>	499893	49.99%
<9, 11, 16>	500107	50.01%

And similarly with the Kind from Figure 5,

```
mkt> demo 1_000_000 with kind
```

```
...>      (<> 1/15 <-1, -15> 1/10 <-1,-5> 1/2 <0,10>
...>      1/30 <9,12> 2/15 <9,20> 1/6 <9,32>).
```

```
Activated 1000000 FRPs with kind
```

```
,----- 2/30 ----- <-1, -15>
|----- 3/30 ----- <-1, -5>
|----- 15/30 ----- <0, 10>
```

```

<> -|
    |----- 1/30 ----- <9, 12>
    |----- 4/30 ----- <9, 20>
    `----- 5/30 ----- <9, 32>

```

Summary of Output Values

Values	Count	Proportion
<-1, -15>	66903	6.69%
<-1, -5>	99555	9.96%
<0, 10>	500256	50.03%
<9, 12>	33545	3.55%
<9, 20>	133086	13.31%
<9, 32>	166655	16.67%

The market tries to show the weighs with a common denominator to make comparison easier, when the denominators are not too large.

With width bigger than 1, the connection between the weights and the proportions of values we get in the table requires a bit more thought. Using the Kind in Figure 4,

```

mkt> demo 1_000_000 with kind
...>      (<> 1 (<-1> 0.4 <-1, -15> 0.6 <-1, -5>))
...>      3 (<0> 1 <0, 10>)
...>      2 (<9> 1 <9, 12> 4 <9, 20> 5 <9, 32>)).
Activated 1000000 FRPs with kind

```

```

,----- 2/5 ---- <-1, -15>
,----- 1 ---- <-1> -|
|                      `----- 3/5 ---- <-1, -5>
|
<> -+----- 3 ---- <0> -+----- 1 ----- <0, 10>
|
|                      ,----- 1 ----- <9, 12>
`----- 2 ---- <9> -+----- 4 ----- <9, 20>
                      `----- 5 ----- <9, 32>

```

Summary of Output Values

Values	Count	Proportion
<-1, -15>	67182	6.72%
<-1, -5>	100190	10.02%
<0, 10>	499985	50.00%
<9, 12>	33438	3.34%
<9, 20>	133213	13.32%
<9, 32>	165992	16.60%

we see that these proportions are essentially the same as what we got above for the Kind in Figure 5, consistent with our claimed equivalence of the two Kinds. Try varying the weights at different stages to get a feel for the relationship between the weights of a higher-width Kind and the demo proportions.

Our work in this subsection provided support for the following claim.⁹

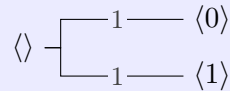
Let K be a Kind of width 1 with weights w_1, w_2, \dots, w_s and corresponding values v_1, v_2, \dots, v_s , and let K' be the Kind (of width 1) with the same values and corresponding weights w'_1, w'_2, \dots, w'_s where

$$w'_i = cw_i, \quad (1.2)$$

for all $i \in [1..s]$ and some $c > 0$. Then, we cannot distinguish a demo of n FRPs with Kind K from a demo of n FRPs with Kind K' .

This claim means that two Kinds whose weights differ by a constant multiplicative factor are in practice interchangeable. We will define a formal notion of Kind equivalence in Section 3.

Puzzle 8. We have seen empirically that a demo of n FRPs with Kind



⁹The *increment* $[1..s]$ is the set of integers from 1 to s , as described in Section F.1.2.

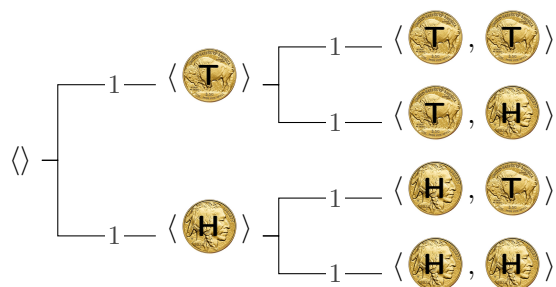
will produce roughly even proportion of 0's and 1's. Let p_1, p_2, \dots, p_k be the proportion of 1s in k repeated demos of n FRPs with this Kind. These are all *approximately* 50%, but they vary around it by a bit from the randomness in the FRPs. How does the variation around 50% depend on n ?



To investigate this, start with several demos of 100 FRPs of this Kind. About how close are is the frequency of 1's to 50%? Look at this as you increase n several times. How big does n need to be so that the frequencies are within about one decimal point of 50%? Within two decimal points? Within three? What can you conclude?



1.4 Kinds as Models

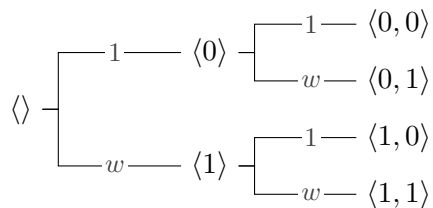
Having seen what FRPs are and gotten some sense of what their Kinds mean, we now turn to the story of how we use them. We will develop this story fully throughout this chapter, and here we lay the groundwork.

Let's start with a simple example, flips of a coin, the first refuge of the probability theorist. We have a random process of interest: we will flip a coin twice in succession. We have a question about this process whose answer we would like to predict: how many times does heads come up? First, the data we measure from this process is an ordered pair listing the results of the successive flips, and we build an FRP to represent these data. This FRP has a Kind that looks like:

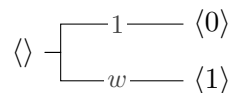


Here the tree describes how the process unfolds in stages. We make the first flip, getting either tails  or heads . Then we flip again, collecting the results in order with four possibilities. Based on what we saw in the last subsection, the weights here tell us that in a demo of many of these FRPs we will get approximately an even proportion of heads and tails. Thus, the FRP represents the data we get from observing the system, and its Kind is a *model* for the system.

While it is fine to have FRPs with arbitrary values (like H or T or even pictures of coins), it is usually more convenient to encode all the outcomes of the system as *numbers*. Sometimes, as when the data itself is a numeric measurement, there is a “natural” way to do this; sometimes, as with the coins, it is arbitrary. Here, for instance, we would typically assign the value 0 to tails  and the value 1 to heads . We also want to allow models of coins for which heads and tails do not show up in even proportions when the coin is flipped. With these tweaks, the Kind for the FRP becomes



in two stages: the first coin flipped and then the second. The same weights are used for every flip, reflecting an assumption that the flips are interchangeable. Indeed, we can see that the Kind of the data FRP is actually built by combining two simpler FRPs – one for each flip – both of Kind



The Kind for the data is thus specified by *assumptions* about the Kinds for the component FRPs representing the individual flips and how those FRPs are combined. We call this set of assumptions a *model*. Finally, we have the question motivating our analysis. We want to predict the *number of heads* in the two flips. From the data we collect, namely the outcome of the two flips, we can compute the number of heads. The function that maps the data we observe to the value that addresses our question (i.e., the number of heads) is called a **statistic**.¹⁰ By mapping the output of the data FRP with this statistic, we get a *new* FRP that describes the particular feature of the data captured by the statistic. The value of this “feature” FRP represents an answer to our question, and it is this value we want to predict. Importantly: we do not know its value until we push the button on the data FRP, and our prediction lets us act *before* we see the data. For reasons we will see, this FRP has Kind:

¹⁰Statistics are discussed in detail in Section 2.

$$\langle \rangle \begin{cases} 1 & \langle 0 \rangle \\ 2w & \langle 1 \rangle \\ w^2 & \langle 2 \rangle \end{cases}$$

and what we will call the “risk-neutral” price for this FRP is

$$\frac{0 \cdot 1 + 1 \cdot 2w + 2 \cdot w^2}{1 + 2w + w^2} = 2 \frac{w}{1 + w},$$

which is our prediction of its value in a sense to be discussed.

If this seems like a lot of work to handle two coin flips, worry not; we will streamline these steps significantly. Still, this breakdown exposes the key pieces in modeling random systems as illustrated in Figure 6.

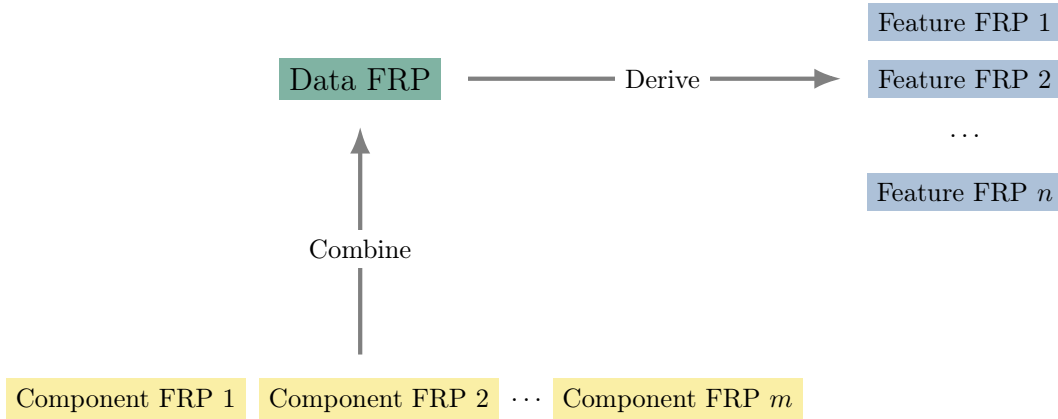


FIGURE 6. Schematic of how FRPs are used. The value of the data FRP represents all the data we measure or observe from a random system or process. This is typically built by *combining* simpler component FRPs whose Kinds and interaction are determined by our knowledge of and assumptions about the system/process. From the data, we *derive* feature FRPs whose values represent features of the data that answer the questions motivating our analysis and are the target of our predictions.

We build an FRP whose value represents the observed output or measured data from a random process or system whose behavior we want to understand or predict. Call this the *data FRP*. For a process of any complexity, we build the data FRP by combining *component* FRPs that represent parts of the random process or system that are easier to understand and reason about. We have questions motivating our study of the random process and particular features of the data that we would like to

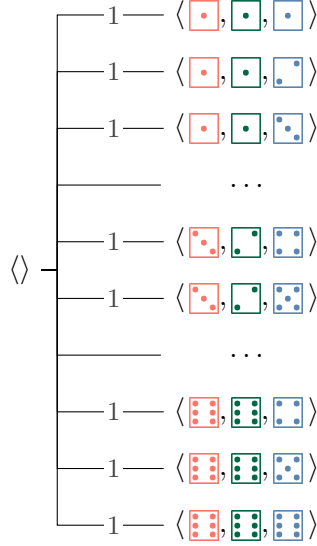





FIGURE 7. The Kind of the FRP representing a roll of three, distinct, balanced dice.

explain and predict. To this end, we derive from the data FRP, one or more *feature* FRPs whose values represent the features of the data that answer our questions. These are the FRPs whose values we most want to predict, and predictions of these values will guide our decisions or actions.

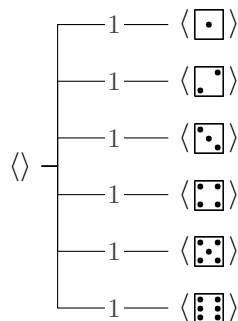
The Kind of the data FRP determines the Kinds of the feature FRPs and thus determine our predictions. The Kind of the data FRP is determined by the Kinds of the component FRPs from which it is built, and the Kinds of the component FRPs – which represent more understandable components of the random process or system – are selected based on our knowledge of and assumptions about the system/process. In that way, the Kinds and predictions that we use to answer our questions are based on a **model**, a collection of assumptions about the Kinds of the component FRPs and the relationships through which they are combined and interact.

Let’s look at two other classic examples. In each of these, we will do some simple computations using the **frp playground** for the Combine and Derive steps from Figure 6 while eliding over the details of how those computations work until the following sections.

For the first example, suppose we have three balanced, six-sided dice   , one red, one green, and one blue. We shake the dice together and roll The data we

observe from this process are the numbers on the three dice, where we distinguish the individual dice. The FRP representing these data has size 216, with values all possible patterns of three rolls. Because the dice are balanced, the Kind of this FRP looks like that shown in Figure 7.

We can see directly that the value of the FRP is composed from the values of three FRPs, one for each of the three dice. Each component FRPs has Kind



As is our habit and preference, we encode the values of these FRPs as numbers, and here the mapping to numbers is direct – the number showing up on the dice. The data then are three-dimensional tuples like $\langle 1, 1, 1 \rangle$ for $\langle \text{red}, \text{green}, \text{blue} \rangle$ and $\langle 3, 2, 4 \rangle$ for $\langle \text{red}, \text{green}, \text{blue} \rangle$.

Enter the playground by invoking `frp playground` at the terminal prompt, and follow along.¹¹ We start by constructing and viewing the Kind of a single roll, the components from which the data are built. Here, `uniform` returns a Kind with equal weights on the specified values.

```
pgd> single_roll_kind = uniform(1, 2, ..., 6)
pgd> single_roll_kind    # Will print the Kind above, output omitted
```

We then construct the FRPs for the three individual dice and combine them into the data FRP `Roll` using the playground's `*` operator, which is what we will call the **independent mixture**¹² of `R`, `G`, and `B`.

```
pgd> R = frp(single_roll_kind)
pgd> G = frp(single_roll_kind)
pgd> B = frp(single_roll_kind)
pgd> Roll = R * G * B
```

¹¹When showing playground input, text from # to the end of a line is a comment for your benefit. You should not type or enter that. Playground output is sometimes omitted after commands to indicate that you should try the command yourself.

¹²Mixtures are discussed in detail in Section 4.

You can examine the Kind of this FRP

```
pgd> kind(Roll)
```

This is the Kind in Figure 7 with the dice rolls encoded as numbers.

Suppose we want to predict (i) the sum of the numbers on the three dice, and (ii) if we see a sum bigger than 10, the minimum of the numbers on the three dice. Notice that the second prediction target is *conditional*: it applies only when the particular condition that the sum of dice is bigger than 10 is true. We need to build the feature FRPs from the data FRP to express these quantities *before we see the value of the data*, so our predictions of these values can guide our actions and decisions.

The playground pre-defines a variety of statistics, functions that take values (tuples of numbers) as input and return to values (tuples of numbers) as output. In particular, **Sum** and **Min** return the one-dimensional tuples (scalar) that compute the sum and minimum of their input tuples:

```
pgd> Sum(1, 2, 3)
<6>
pgd> Min(4, 7, 2)
<2>
```

Transforming the FRP **Roll** with the statistic **Sum** gives a new FRP whose value is the sum of the components in **Roll**'s value. In the playground, we write this as

```
pgd> DiceSum = Roll ~ Sum
```

where we think of \sim as an arrow connoting that the value of **Roll** is passed through the statistic **Sum** to produce the new FRP **DiceSum**.

Our predictions of the sum of the dice are determined by `kind(DiceSum)`, and we compute our prediction of its value – its “risk-neutral price” or **expectation**¹³ – with the **E** operator:

```
pgd> kind(DiceSum)
,---- 0.0046296 ---- 3
|---- 0.013889 ---- 4
|---- 0.027778 ---- 5
|---- 0.046296 ---- 6
```

¹³Both terms are discussed in detail in Section 7.

```

      |---- 0.069444 ----- 7
      |---- 0.097222 ----- 8
      |---- 0.11574 ----- 9
      |---- 0.12500 ----- 10
<> -|
      |---- 0.12500 ----- 11
      |---- 0.11574 ----- 12
      |---- 0.097222 ----- 13
      |---- 0.069444 ----- 14
      |---- 0.046296 ----- 15
      |---- 0.027778 ----- 16
      |---- 0.013889 ----- 17
      `---- 0.0046296 ---- 18
pgd> E(DiceSum)
21/2

```

We can see from the Kind that the weights are *symmetric* around the midpoint of the possible values, and the expectation is equal to that midpoint: 10.5.

Look also at the values of the FRPs themselves to see how Roll's value is built from R, G, and B and how DiceSum's value is derived from Roll's.

```

pgd> R
An FRP with value <4>
pgd> G
An FRP with value <6>
pgd> B
An FRP with value <6>
pgd> Roll
An FRP with value <4, 6, 6>
pgd> DiceSum
An FRP with value <16>

```

These are the values for the FRP that I drew from the Warehouse; your values will likely be different.

We can draw new FRPs from the Warehouse of the same Kind as any FRP we have using the `clone` function. (You may not get the same values in your output

because you are obtaining a different FRP with the same Kind.) We can also draw multiple such copies (like a demo in the market) with the `FRP.sample` function.

```
pgd> clone(Roll)
An FRP with value <6, 3, 2>
pgd> FRP.sample(10000, DiceSum)
|-----+-----+-----|
| Values | Count | Proportion |
|-----+-----+-----|
| 3      | 41    | 0.41%      |
| 4      | 135   | 1.35%      |
| 5      | 285   | 2.85%      |
| 6      | 431   | 4.31%      |
| 7      | 661   | 6.61%      |
| 8      | 938   | 9.38%      |
| 9      | 1198  | 11.98%     |
| 10     | 1265  | 12.65%     |
| 11     | 1267  | 12.67%     |
| 12     | 1145  | 11.45%     |
| 13     | 1027  | 10.27%     |
| 14     | 712   | 7.12%      |
| 15     | 452   | 4.52%      |
| 16     | 279   | 2.79%      |
| 17     | 116   | 1.16%      |
| 18     | 48    | 0.48%      |
|-----+-----+-----|
```

Compare these proportions with the Kind of `DiceSum` above, and try several (perhaps larger) demos with `FRP.sample`.

For prediction target (ii), we want to compute the minimum of `DiceSum`'s components but only on the condition that the sum is bigger than 10. For this, we use a **conditional constraint**.¹⁴ In the playground, we specify a conditional constraint with a `|` which is read as “given.” We express our constraint with

```
pgd> Roll | (Sum(__) > 10)
```

¹⁴Conditional constraints are discussed in detail in Section 5.

where $(\text{Sum}(__) > 10)$ is the condition (the parentheses are needed) and $__$ stands for the value of the FRP on the left side of $|$. What you see when you enter this will depend on whether your copy of `Roll` has value with sum bigger than 10. If it does, you will see the same value as that of `Roll`; if not, you will see the empty FRP. Try repeating `clone(Roll) | (Sum(__) > 10)` until you see both possibilities. Now, for `Roll` – or a clone with sum bigger than 10 –

```
pgd> kind(Roll | (Sum(\_\_) > 10))
```

and observe that the leaf nodes all satisfy the condition, i.e., all the values have sum bigger than 10. Our feature FRP for (ii) is

```
pgd> M10 = Min(Roll | (Sum(\_\_) > 10))
pgd> M10
pgd> kind(M10)
pgd> E(M10)
2.722
```

(Again, it is more interesting to use a clone of `Roll` that satisfies the condition; otherwise, you just get the empty FRP and empty Kind.) Look at this Kind. Try comparing it to `Min(Roll | (Sum(__) > 16))` and `Min(Roll | (Sum(__) > 5))`.

Our second example is the (in)famous Monty Hall game, which goes as follows:

1. You are faced with three doors: left, middle, right.
2. Monty has selected a door at random and placed a prize behind it; the other two doors have nothing behind them.
3. You choose a door.
4. Monty – the MC of our game – opens one of the other doors revealing that it does not hide the prize.
5. He offers you a chance to switch doors.
6. You indicate whether you will switch your choice.
7. Your final door is opened. If you picked the prize, you win; otherwise, you lose.

Should you take Monty's offer to switch?

To begin, consider the *strategies* available to you in this game. Each strategy specifies: i. how you pick your initial door, and ii. whether you accept Monty's offer to switch from your initial door choice. For example, one strategy is (Pick the Left

Door, Do not switch); another is (Pick the Door based on a size 3 FRP with equal weights, Switch). We will analyze each distinct strategy separately, *using one FRP per strategy* to represent the game’s outcome.

Once your strategy has been specified, the data from the game is the sequence of decisions made at each stage. We build an FRP to represent these data. Those decisions are:

1. Monty hides the prize behind the left, middle, or right door.
2. You select either the left, middle, or right door according to your strategy.

These are illustrated in Figure 8.

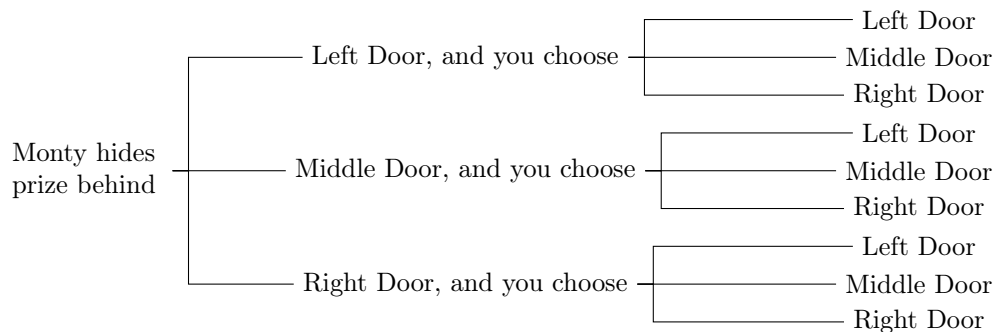


FIGURE 8. The decision tree leading to your initial door choice in the Monty Hall game. For any given strategy, these are the choices that determine the outcome of the game.

We model this process with an FRP. First, we assign numeric values to the outcome at each stage, with Left Door as 1, the Middle Door as 2, and the Right Door as 3. Second, we assign weights based on the description of the problem. Note that Monty has placed the prize behind a door picked at random with equal weight on each door, and then you pick a initial door according to your particular strategy. An FRP reflecting this interpretation thus has Kind shown in Figure 9. Here, we have quantified your strategy as an arbitrary choice of positive weights ℓ, m, r and a choice of whether to switch.

It turns out, as we will see later, that the choice of weights has no impact on our analysis, so we will focus on comparing the “Don’t Switch” and “Switch” strategies.

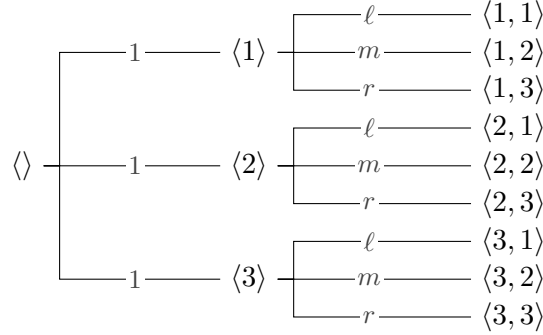


FIGURE 9. The Kind for the FRPs modeling the Monty Hall game. In each value list, the first element is Monty's door choice and the second element is your door choice. The weights ℓ , m , and r are discussed in the text.

Puzzle 9. What does it say about your strategy if ℓ , m , and r are all equal? What does it say about your strategy if $\ell = 1 = r$ but m is very, very, very large?

By the game's structure, if you do *not* switch, then you only win if you chose the prize initially; if you do switch, then you only win if you did *not* choose the prize initially.

Puzzle 10. (Important!)

For each leaf node in Figure 9, fill in the table below indicating whether you Win or Lose under the DON'T SWITCH and the SWITCH strategies.

Value	Don't Switch	Switch
$\langle 1, 1 \rangle$		
$\langle 1, 2 \rangle$		
$\langle 1, 3 \rangle$		
$\langle 2, 1 \rangle$		
$\langle 2, 2 \rangle$		
$\langle 2, 3 \rangle$		
$\langle 3, 1 \rangle$		
$\langle 3, 2 \rangle$		
$\langle 3, 3 \rangle$		

This means that if the FRP's value is denoted by $\langle d_{\text{Monty}}, d_{\text{You}} \rangle$, then

- If you do not switch, you win if and only if $d_{\text{Monty}} = d_{\text{You}}$.
- If you do switch, you win if and only if $d_{\text{Monty}} \neq d_{\text{You}}$.

We can now transform the output of each FRP – one for SWITCH and one for DON'T SWITCH – using a *statistic* that gives a value 1 if you win and 0 if you lose in either case. This gives us new FRPs with Kinds shown in Figure 10 for the no switch case (top) and the switch (bottom).

We will see how to do this in Section 2.

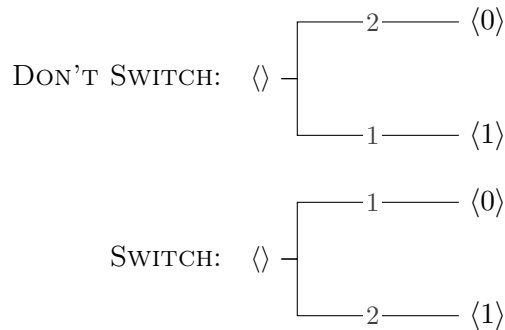


FIGURE 10. The Kind for the transformed FRPs in the no-switch and switch cases, respectively. Notice that the weights in the two cases are different and that they do not depend at all on ℓ , m , or r .

In the playground,¹⁵ we first load the module `frplib.examples.monty_hall` from `frplib`. This imports two pre-defined Kinds and a predefined statistic:

```
pgd> from frplib.examples.monty_hall import (
...>     door_with_prize, chosen_door, got_prize_door_initially
...> )
```

¹⁵Here and later, text from # to the end of a line is a comment for your benefit. You should not type that.

The first two are Kinds, described as follows:

- `door_with_prize` models which door has the prize, giving equal weight to 1, 2, and 3; and
- `chosen_door` models your initial door choice. It has arbitrary weights ℓ, m, r on the three doors.

We can combine these with an independent mixture (the `*` operator) to get the Kind in Figure 9:

```
pgd> game_outcome = door_with_prize * chosen_door # Kind in Fig 9
```

This is the *Kind* of the FRP that represents the measured data in this game. We can build the FRP in the playground that represents a game outcome, but we first need to specify at least our strategy for choice of doors because those are given symbolically in `game_outcome`. For instance:

```
pgd> Game = frp(outcome_by_strategy(left='1/3', middle='1/3', right='1/3'))
```

```
pgd> Game
```

```
An FRP with value <2, 3>
```

where `outcome_by_strategy` returns a version of `game_outcome` with the values of ℓ , m , and r as specified. (We will see from the Kinds that this choice does not actually impact our decision.)

The statistic `got_prize_door_initially` takes a pair $\langle d_{\text{Monty}}, d_{\text{You}} \rangle$, where d_{Monty} is the door with the prize and d_{You} is the door you chose initially, and returns 1 (for true) if $d_{\text{Monty}} = d_{\text{You}}$ or 0 (for false) otherwise. Transforming the data FRP for the game with this statistic gives a feature FRP whose value represents whether you win under the DON'T SWITCH strategy (or lose under the SWITCH strategy).

```
pgd> Game ^ got_prize_door_initially
```

```
An FRP with value <0>
```

which has Kind

```
pgd> dont_switch_win = game_outcome ^ got_prize_door_initially
```

```
pgd> dont_switch_win
```

```
,---- 2/3 ---- 0
```

```
<> -+
```

```
`---- 1/3 ---- 1
```

which is consistent with the DON'T SWITCH Kind in Figure 10 in that losing has twice the weight of winning. Observe that we can transform a *Kind* with a statistic in the same way that we can transform an FRP, and the results are consistent: the Kind of a transformed FRP is the same as the transform of the FRPs Kind. So, `dont_switch_win` equals `kind(Game ^ got_door_prize_initially)`.

From `got_prize_door_initially`, we can define the complementary statistic `didnt_get_prize_door_initially`


```
pgd> didnt_get_door_prize_initially = Not(got_prize_door_initially)
```

This returns 1 when `got_prize_door_initially` returns 0 and vice versa. Transforming `Game` (and its kind) with this statistic gives

```
pgd> Game ^ didnt_get_prize_door_initially
An FRP with value <1>
pgd> switch_win = game_output ^ didnt_get_prize_door_initially
pgd> switch_win
      ,---- 1/3 ---- 0
<>  -+
      `---- 2/3 ---- 1
```

which is consistent with the SWITCH Kind in Figure 10 Notice that both `dont_switch_win` and `switch_win` are independent of your choice of ℓ , m , and r .

We can also activate FRPs of each Kind to simulate the outcome of many games.

```
pgd> FRP.sample( 12_000, dont_switch_win )
Summary of output values:
0          7929 (66.1%)
1          4071 (33.9%)
```

```
pgd> FRP.sample( 12_000, switch_win )
Summary of output values:
0          3954 (32.9%)
1          8046 (67.1%)
```

Here, we pushed the buttons on 12,000 FRPs of each Kind `dont_switch` and `switch`. The results are clear cut: switching is the best choice.

1.5 The Big 3+1!

Almost everything we will do with probability theory – simulation, modeling, inference, decision making, prediction – is built on four principle operations, which we call the **Big 3+1**. Three of these operate on FRPs to produce new FRPs by connecting output ports to input ports in several ways: (1) transforming the output of an FRP by some algorithm (statistics), (2) generating random outcomes contingent on

earlier outcomes (mixtures), and (3) accounting for partially observed information (conditionals). The fourth operation takes an FRP and yields a prediction of the FRP's value (expectation). All these operations on FRPs all have directly analogous operations on Kinds, and all of them are fundamental tools for building and analyzing models of real systems.

1. **Transforming with Statistics** (Section 2)

A *statistic* is just a function that maps values (i.e., tuples of numbers) to values (tuples of numbers). Whenever we want to transform, summarize, or extract a feature from our data, we define a statistic that does the job. Whenever we apply an algorithm to process or analyze our data, we are using a statistic.

Although it is a function from values to values, we can use a statistic to *transform* an FRP or a Kind. We transform an FRP by *applying the statistic to the FRPs value*, producing a new but related FRP. We transform a Kind by *applying the statistic to each possible value of the kind* (i.e., the leaf nodes) and then combining branches that map to the same value in the transformed tree, adding their weights.

We use statistics to **express and answer questions**. The “Derive” step in Figure 6 uses statistics (and conditionals) to build feature FRPs whose values answer our questions. The statistic describes the steps we will take to extract the desired information from the data, before those data are available. Each feature FRP is derived by transforming the FRP representing the data using a statistic that represents one of our questions. The values of these FRPs answer those questions, and our goal is to predict those values (or compute the Kind) as accurately as possible to guide our decisions and actions.

2. **Building with Mixtures** (Section 4)

We use mixtures **to build a model by combining simpler components**. When a system can most easily be described in terms of its parts and how they combine and interact, think about using a mixture. The “Combine” step in Figure 6 uses mixtures (and statistics) to build the FRP for the data we measure from FRPs that represent the simpler parts of the process or system. In the Monty Hall example, we have seen one type of mixture, where the different stages of the process do not interact (Monty chooses a door, you choose

a door). This is called an independent mixture. More generally, though, the different components will interact. For instance, what happens at one stage will influence the outcomes of later stages. A general mixture reflects this by passing the values produced at one stage into the next and collecting that entire history.

3. Constraining with Conditionals (Section 5)

We use conditionals to **update our knowledge and predictions with new information**. A conditional is a constraint telling us that some specific observable condition is *known to be true*, either because we directly observed that condition or because we are considering the hypothetical in which we observe it.

When we constrain a Kind with a conditional, we simply *erase all the branches that are inconsistent with the condition*. This gives us a new Kind, which in canonical form simply re-normalizes the weights of the remaining branches by the total weight of branches that are consistent with the condition.

If you want to update your knowledge or predictions for some known information, use a conditional.

4. Predicting with Expectations (Section 7)

We use the risk-neutral price **to compute our best prediction of an FRP's value**, which we call its **expectation**. The expectation of an FRP reflects a “typical” value that is “close” in some sense to what the FRP will produce.

From their definition as risk-neutral prices, expectations inherit many useful properties, and from these properties, we can deduce how to compute the expectation of an FRP from its Kind. Computing expectations is often the target of our analysis because predictions can guide our behavior and decisions in the context of the system we are studying.

By understanding how the Big 3+1 operate – transforming values, erasing branches, combining stages, weighted averages – we can recognize and exploit these operations even in more complicated calculations and contexts.

Checkpoints

After reading this section you should be able to:

- Describe what an FRP is and what each of the words fixed, random, and payoff in the name refers to.
- Explain the structure of a valid FRP *Kind*.
- Invoke `frp market` and `frp playground` from the terminal prompt and get basic help.
- Use the `frp market` to examine the values of FRPs of a given Kind.
- Guess to within a reasonable margin of error the proportions of each value observed in a large demo of FRPs of a specified Kind.
- Explain in rough terms what the weights on an FRP's Kind indicate.
- Explain in broad strokes how FRPs might be useful for modeling real systems.
- List the Big 3+1 operations on FRPs and Kinds.

2 Transforming with Statistics

Key Take Aways

The data we collect from observing random processes is often high dimensional, but our interest is usually in specific information derived from that data. A **statistic** is a data processing algorithm, a function that takes tuples as input and returns tuples as output. A statistic that accepts tuples of dimension n as input and returns tuples of dimension n' as output has **type** $n \rightarrow n'$, **codimension** n , and **dimension** n' .

An FRP is transformed with a statistic by connecting the All output port of the FRP to the input port of an empty FRP with an adapter that represents the statistic. When the FRP is activated, its value is passed to the adapter which applies the statistic to the value and activates the empty FRP with the result as its value. This produces a new FRP whose value is obtained by *applying the statistic to the value of the original FRP*.

We can also transform a *Kind* with a statistic: **we first apply the statistic to each value (leaf node) of the Kind and then combine branches that map to the same value, adding their weights**. For any compatible statistic, the Kind of a transformed FRP is the transformed Kind of the FRP.

When we make observations or collect data of any sort for learning or discovery, we usually *do* something with it – process it, analyze it, summarize it. We might transform the data into a more useful form, compute summaries to help us understand it, or extract features that answer our questions. The data-processing algorithms that do this are collectively called statistics. A *statistic* is a function that takes a value (tuple of numbers) as input and returns a value (tuple of numbers) as output.

An FRP represents data measured from some random system, and we can apply a statistic to its value *once it is activated* to produce a transformed value. The result is a *transformed FRP* whose value is the output from the statistic when given the original FRP's value as input.

We create a transformed FRP by connecting the All output port of the original FRP to the input port of an empty FRP using an adapter like that shown in Figure 11. The adapter has circuitry to compute a specific statistic with the value of the

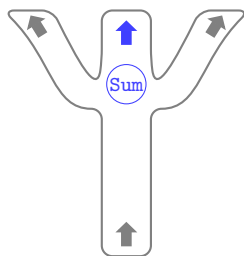


FIGURE 11. An adapter used to transform an FRP by the Sum statistic in the playground, which computes the sum of its input's components. The value comes from the original FRPs All output port through the bottom port on the adapter, is transformed by the statistic, and is emitted through the top center port. The other two ports on top simply copy the original value, allowing us to construct multiple transforms or mixtures at the same time. Different statistics' adapters look the same but have different names and internal circuitry. That the adapter resembles the letter ψ will be a useful mnemonic.

original FRP, when it is produced, as input. The original FRP's output port is connected to the bottom of the adapter and the transformed value is emitted from the central output port on top, which is connected to the **Input** port of an empty FRP. (The adapter's other two output ports simply copy its input so that we can create multiple transforms of the same original FRP.) When this connection is made, the empty FRP is automatically reconfigured: its output ports are relabeled accordingly, and its Kind display recomputed. The transformed FRP will be activated when the original FRP is, making the original data available to the statistic. This process only works with an adapter for a statistic that is *compatible* with the original FRP in the sense that the statistic can accept all possible values produced by the original FRP.

In this section, we consider in detail the operation of transforming an FRP and its Kind with a statistic. We will formally define what statistics are, their key properties, and how we specify them. With many examples, we will illustrate how we use statistics to express and answer questions, to transform FRPs and Kinds. And in the playground, we will explore the many built-in statistics along with how to create custom statistics, and we will learn how to build transformed FRPs and Kinds.

2.1 Statistics and Data Processing

Consider an FRP that represents five flips of a balanced coin. The values of this FRP are 5-tuples like $\langle 1, 0, 0, 1, 1 \rangle$ or $\langle 0, 1, 0, 0, 1 \rangle$, where 0 and 1 stand for tails and heads.

Suppose we are interested in the number of heads in those five flips. To answer this question, we use the statistic **Sum** that takes a tuple as input and returns the sum of its components. $\text{Sum}(\langle 1, 0, 0, 1, 1 \rangle) = 3$ and $\text{Sum}(\langle 0, 1, 0, 0, 1 \rangle) = 2$. Transforming this FRP by **Sum** gives a new FRP with possible values 0, 1, 2, 3, 4, and 5. But these two FRPs are connected. For instance, when the original FRP produces value $\langle 1, 0, 0, 1, 1 \rangle$, the transformed FRP produces value 3.

Puzzle 11. What are two more questions we might want to ask about this sequence of five coin flips? Describe statistics that answer these questions. They should accept a 5-tuple as input and return a tuple that answers the corresponding question for the input.

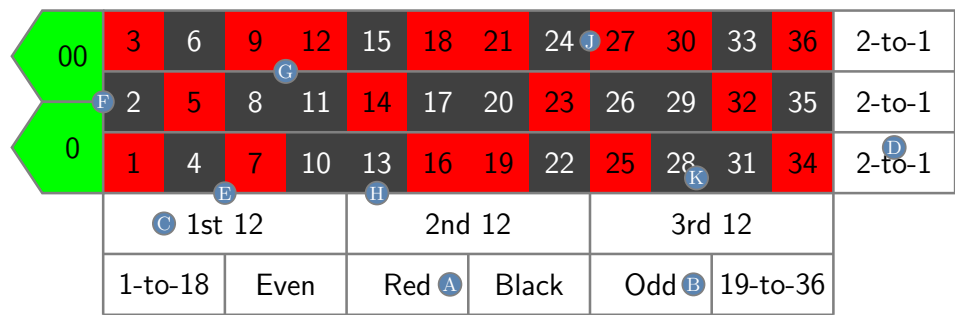


FIGURE 12. Betting table for roulette. The pocket numbers are oriented with top to the left, so what are called “columns” go from left to right. The labeled disks represent chip placements corresponding to plays in Table 1.

Example 2.1 Roulette

A Nevada roulette wheel has 38 pockets along its edge that can catch and hold a small metal ball. Each pocket has a color and a unique number: two pockets are colored green with numbers 0 and 00 and the remaining pockets are numbered 1 through 36, with half colored black and half colored red.

A game of roulette begins with players making their bets by placing chips worth some amount of money on a betting table that displays an array of numbered, colored squares and surrounding, labeled tabs, as shown in Figure 12. Each chip placed specifies a *set of pockets* (based on where the chip is placed)

Play	Winning Pockets	Payoff	Sample
Even Money	A set of eighteen numbers: red, black, even, odd, 1–18, 19–36	1 to 1	A, B
Dozen	One of 1–12, 13–24, 25–36	2 to 1	C
Column	Twelve numbers in one “column”	2 to 1	D
Six Line	Six consecutive numbers (two “rows”)	5 to 1	E
Top Line	00, 0, 1, 2, 3	6 to 1	F
Corner	Four numbers that share a corner	8 to 1	G
Street	Three consecutive numbers (one “row”)	11 to 1	H
Split	A pair of adjacent numbers	17 to 1	J
Straight	A single number	35 to 1	K

TABLE 1. Common bets in roulette. Each such bet wins when the ball lands in a particular set of pockets. If a player loses a play, the amount bet is forfeit to the “house.” If the player wins, then the casino returns the bet plus a payoff that is a multiple of the amount bet. The last column gives the labels of chip placements in Figure 12 that exemplify the play.

and an amount bet (based on the value of the chip). A roulette bet is called a “play” in the official lingo. The players bet and then the ball is released into the spinning wheel. The ball rolls around the wheel and eventually comes to rest in one of the pockets. A play whose set of pockets includes the one with the ball wins; the player keeps the amount bet and receives a payoff that is a multiple of the amount bet, depending on the play. A play whose set of pockets does *not* include the one with the ball loses, and the player forfeits the amount bet.

On the betting table in Figure 12, squares numbered 1 through 36 are colored red or black and the tabs 0 and 00 are colored green, matching the wheel. The remaining regions specify a set of pockets like “Red” for all the red pockets or “2nd 12” for pockets 13–24. The basic plays are described in Table 1 in order of increasing payoff. For instance, a \$1 Top Line play will win \$6 if the ball stops in pockets 00, 0, 1, 2, or 3 or will lose \$1 otherwise. As the number of winning pockets for a play decreases, the payoff increases. The Sample column in the table indicates which chip placements in Figure 12 match that play.

In the playground, load the example code from `frplib` as follows:


```
pgd> from frplib.examples.roulette import roulette
```

This imports an object `roulette` that has everything we will need for this example. First, calling `roulette` as a function with no arguments returns a fresh FRP representing a single spin of the roulette wheel.

```
pgd> roulette()
```

```
An FRP with value <21>
```

As usual, we assign a number to each of the possible values, using the pocket number except for pocket 00 to which we assign the value -1. The Kind of this FRP – the Kind for a single spin of the wheel – is available as `roulette.kind`.

```
pgd> roulette.kind
```

Looking at the weights, this Kind assumes that the wheel is symmetric, with no preference given to any pocket over another.

To model a bet on a single spin of the wheel, we build and name one such FRP:

```
pgd> R = roulette()
```

This FRP is fresh, and before we activate it (by examining its value), its value – the pocket in which the ball stops – is uncertain. `R` represents the data we measure for a single spin, all the outcomes we care about in the game are derived from it. So `R` is what we called the “data FRP” in the previous section. We can use the function `Kind.equal` to check that `roulette.kind` is the same as `kind(R)`:

```
pgd> Kind.equal(roulette.kind, kind(R))
```

```
True
```

The questions driving our analysis are not about the value of `R` itself but about the outcome of various plays. What is our return from a bet on Red? On Top Line? Which of the standard plays, if any, is best? Worst? (We might also wonder what our best betting strategy is for choosing plays on multiple spins, a question we take up later.) We can represent each such question with a *statistic*

that takes as input a value of `R` and returns an answer. For instance, all of the standard plays are available through the `roulette` object in the playground, e.g.,

```
pgd> roulette.red
A Statistic 'red' that expects 1 argument (or a tuple of that
dimension) and returns a scalar.
pgd> roulette.top_line
A Statistic 'top_line' that expects 1 argument (or a tuple of that
dimension) and returns a scalar.
pgd> roulette.straight(16) # 16 is the number you're betting on
A Statistic 'straight_16' that expects 1 argument (or a tuple of that
dimension) and returns a scalar.
```

All of these act as functions that take a pocket number and return the value of a \$1 bet if the ball stops in the given pocket.

```
pgd> roulette.red(3)
<1>
pgd> roulette.red(4)
<-1>
pgd> roulette.top_line(-1)
<6>
pgd> roulette.top_line(17)
<-1>
pgd> s16 = roulette.straight(16)
pgd> s16(16)
<35>
pgd> s16(17)
<-1>
```

For the straight play, we could call it directly `roulette.straight(16)(17)`, but for clarity, we instead name and store the statistic in a variable `s16` and use that.

If we are interested in a bet of a different amount, we can scale the statistics. When we multiply a statistic by a number, we get a new statistic. For example:

```

pgd> r25 = 25 * roulette.red
pgd> s16_10 = 10 * s16 # == 10 * roulette.straight(16)
pgd> r25(3)
<25>
pgd> r25(4)
<-25>
pgd> s16_10(16)
<350>
pgd> s16_10(17)
<-10>

```

Notice that $10 * s16(16)$ is the same *value* as $(10 * s16)(16)$ and as $s16_10(16)$, but $s16(16)$ is a value that we scale, whereas $10 * s16$ is a *statistic* that we evaluate at 16.

The statistic `s16_10` encapsulates the question: what is the return of a \$10 straight play on pocket 16. We can use this statistic to transform the FRP `R`.

```
pgd> W16_10 = R ~ s16_10
```

The value of this FRP is the return on a \$10 straight play on 16 with the spin represented by the value of `R`. Notice how the values of `R` and `W16_10` are connected by the statistic `s16_10`.

```

pgd> R
An FRP with value <23>
pgd> W16_10
An FRP with value <-10>
pgd> s16_10(23)
-10

```

Note that in the playground, there are two equivalent and interchangeable ways to transform an FRP:

```

pgd> W16_10 = R ~ s16_10
pgd> W16_10 = s16_10(R)

```

The second matches the mathematical notation we will use downstream; it captures the idea that we are taking the *value* of R as input to the statistic. Although `s16_10(R)` looks like we are calling the function with R as the argument, that is just a metaphor for a higher-level operation.

For each of the transformed FRPs representing the value of a play, we can compute its Kind, e.g.,

```
pgd> kind(W16_10)
      ,---- 37/38 ---- -10
<> -|
      `---- 1/38 ----- 350
```

How much is this FRP worth? We compute its “risk-neutral price” with the `E` operator:

```
pgd> E(W16_10)
-0.5263157894736842
```

which equals $-10/19$. Thus, our predictions tell us that owning this FRP is a loss; someone would have to *pay you* \$10/19 (about 53 cents) to accept this.

Puzzle 12. Compute a transformed FRP for one of each of the standard plays listed in Table 1, look at its Kind and expectation. What do you conclude?

Note that for plays like Column and Corner, we specify the particular play by giving the lowest numbered pocket. For example, the following are statistics for a \$1 play: `roulette.column(2)` for the middle column starting with pocket 2, `roulette.corner(8)` for the four pockets 8, 9, 11, 12. For Split, you need to specify the adjacent positions in order, e.g., `roulette.split(24,27)` or `roulette.split(23,24)`. Call `help(roulette.split)` etc. for details.

What if we want to model a bet that combines multiple standard plays? For instance, a combined \$10 even play, a \$5 corner play on (25, 26, 28, 29), a \$20 straight play on 4, and a \$50 column play on the second column corresponds to the following statistic:

```
pgd> comb = 10 * roulette.even + 5 * roulette.corner(25) +
           20 * roulette.straight(4) + 50 * roulette.column(2)
```

```
pgd> comb(26)
<130>
```

The corresponding feature FRP is `comb(R)`. Look at its value and relate it to your value of `R`. It has Kind

```
pgd> kind(comb(R))
,---- 13/38 ---- -85
|---- 10/38 ---- -65
|---- 1/38 ----- -40
|---- 1/38 ----- -20
<> -+---- 5/38 ----- 65
|---- 5/38 ----- 85
|---- 1/38 ----- 110
|---- 1/38 ----- 130
`---- 1/38 ----- 655
```

and risk-neutral price $E(\text{comb}(R)) \approx -4.47$, so we predict that on average you would lose about \$4.47 per attempt on this combined play.

The previous example is illustrative in several ways. First, it shows how we observe data from a system and extract information from it to answer our questions. Each statistic, like `roulette.even` or `roulette.straight(16)`, takes the data as input and computes an answer to one question.

Second, it shows how we do our analysis while the FRPs are still fresh. If we *had* the data in hand, we could simply apply the statistic to compute an answer, and there would be nothing to predict. This is why we transform the data FRP. A feature FRP like `roulette.even(R)` is only activated when `R` is activated, but we can compute its Kind and thus predict its value. If we are choosing among various plays, it does us no good to answer the question after betting is closed. These predictions are made before all the data is observed – and the answers to our questions determined – so that we can make decisions or take actions before it is too late.

Third, it reveals how our model drives our conclusions. A model is a collection of assumptions about the system under study. For roulette, our model is that a spin of the wheel gives no preference to any pocket over any other. This assumption is empirically checkable, and it is aligned with casinos' incentives, else gamblers could

seek out and exploit any systematic deviation from symmetry.¹⁶ The assumption gives us the Kind of R which in turn determines the Kind of the feature FRPs like `roulette.even(R)`. If the assumption were poor, we could update our model accordingly with no other changes in our procedure. Why not just focus on modeling the outcome of a particular bet? One reason is that we often have multiple questions to answer. A more important reason is that we usually understand better how to model the data – or the components that constitute it – than we do the derived features.

¹⁶This has indeed happened.

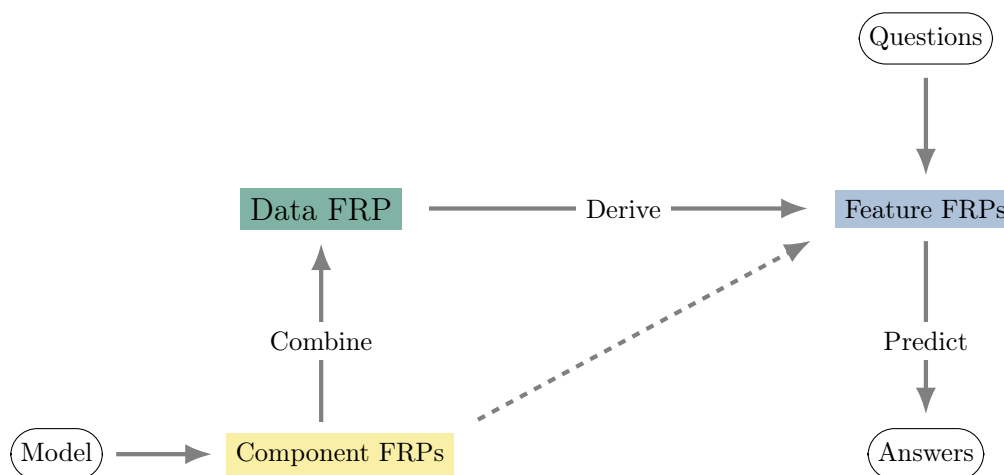


FIGURE 13. Update to Figure 6 that shows the role of the model, motivating questions, and predictions of their answers. The dashed arrow indicates that some feature FRPs may be defined in terms of selected components and thus may be activated before the data FRP is.

Figure 13 updates the schematic in Figure 6 to show the role of the model, our motivating questions, and prediction of the answers to those questions. The Model is a collection of assumptions that we use to build the Component FRPs by specifying their Kinds. Using statistics and mixtures, we Combine the components into an FRP that represents all the data that will be observed from the system, the Data FRP. The Questions we want to answer with these data focus our attention on specific features of the data, and we transform the Data FRP with statistics (along with conditionals) to build the FRPs that represent these features. In many cases, we observe partial information about the features as the system evolves. The dashed arrow in the Figure indicates that some Feature FRPs may also be expressed as

transforms of selected Component FRPs and thus can activate before the Data FRP does. We then use expectations to predict the values of the Feature FRPs and so predict the Answers to our Questions.

The Roulette example illustrates most of these pieces, except the data are simple enough that there is only a single Component FRP, which equals the Data FRP. Our model is that all 38 pockets have an equal weight, from which we derive the component's Kind. Our questions center on the performance of various plays, so our statistics map the pocket number to the return on those plays. The Feature FRPs represent the value of those plays *on the actual bet*, and we predict their value by finding their Kind and expectation ("risk-neutral price").

If instead we had studied *two* spins of the Roulette wheel, we would use our earlier model to describe each spin's Kind but would extend the model with an assumption about how the two spins are related. (Does the first spin give you information to predict the second spin?) There would be two Component FRPs, one per spin, and our questions would focus on overall performance for *pairs* of plays on the two spins (including not betting). With even more spins, there are more components (one per spin), more interactions among them, more possible betting strategies reflected in a broader range of statistics and Feature FRPs to consider.

Definition 1. A **statistic** is a function that takes *values* in some set as inputs and returns a *value* as output. We require that the dimension of the output value depends only on the dimension of the input value.

If a statistic accepts values of dimension n and given such returns a value of dimension n' , we say that the statistic has **type** $n \rightarrow n'$. We call n the **codimension*** and n' the **dimension**. If a statistic has dimension 1 for all valid inputs, we say that it is a *scalar statistic*. It is possible for a statistic to have more than one distinct type if it can accept input tuples of various lengths, but a statistic must have at most one type for each codimension.

We will typically use Greek letters to denote statistics, especially ψ ("psi", pronounced like sigh), φ ("phi" pronounced fee or fi), ξ ("xi", pronounced zigh or ksee), and ζ ("zeta"). Some special statistics may be given meaningful names.

*Pronounced "ko-dimension".
Some call this *arity*.

Here, the word *value* has a specific meaning: a *tuple* (aka list). We almost always use numeric values – tuples of numbers – but from time to time we will consider

alternative types, like Booleans.¹⁷ If a value is a list of dimension 1, we elide any distinction between the tuple and the item it holds.

A particular statistic may or may not have a single unique codimension. For example, a statistic that takes a point in the plane $\langle x, y \rangle$ and returns its distance from the origin $\sqrt{x^2 + y^2}$ has codimension 2 and dimension 1, so is of type $2 \rightarrow 1$. A statistic that maps two points in space $\langle x_1, y_1, z_1, x_2, y_2, z_2 \rangle$, encoded as a 6-tuple, to the midpoint between them $\langle \frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}, \frac{z_1+z_2}{2} \rangle$ has type $6 \rightarrow 3$. However, many statistics naturally accept tuples of various lengths. For instance, the statistic that reverses a list, mapping $\langle x_1, x_2, \dots, x_n \rangle$ to $\langle x_n, x_{n-1}, \dots, x_1 \rangle$, has type $n \rightarrow n$ for every integer $n \geq 0$, and the statistic that computes the maximum of the components, mapping $\langle x_1, x_2, \dots, x_n \rangle$ to $\max(x_1, x_2, \dots, x_n)$, has type $n \rightarrow 1$ for every integer $n \geq 0$.¹⁸

¹⁷To avoid repeating this qualification, we will often focus on the leading case in the text and refer to values as tuples of numbers.

¹⁸We define the maximum of an empty tuple to be $-\infty$.

Notational Convention. For a function ψ that takes an n -tuple as input, it is convenient to be flexible with how we write its arguments when evaluating the function.

If $v = \langle v_1, \dots, v_n \rangle$, we treat the following as equivalent and interchangeable:

$$\psi(\langle v_1, \dots, v_n \rangle) \qquad \psi(v) \qquad \psi(v_1, \dots, v_n)$$

using whichever form is clearest and most convenient at any moment. This is discussed in detail in Section F.7 of Interlude F.

When a statistic with codimension n is given as input an n -tuple, we require that it always return a tuple of some common dimension n' . This ensures that all the values of an FRP of dimension n , when passed to the statistic, give tuples of a fixed dimension, which can be the output of an FRP of dimension n' . Sometimes in practice, we meet this requirement by “padding” the output tuple with items that bring it to the right length without changing our interpretation of the value.

Definition 2. An FRP X and a statistic ψ are **compatible** if every possible value of X is a valid input to ψ .

This implies that the dimension of X is equal to a codimension of ψ and that whatever value X produces can be passed as input to ψ .

If an FRP and a statistic are compatible, it means that we can attach the All output port of the FRP to the statistic’s adapter like that illustrated in Figure 11. (If they are not compatible, the connecting cables will not fit, making attachment physically impossible.) We then plug the output of the adapter to an empty FRP to create a new *transformed* FRP.

Definition 3. If X is an FRP and ψ is a compatible statistic, then X transformed by ψ – denoted by $\psi(X)$ – is the FRP that produces value $\psi(\langle v_1, \dots, v_n \rangle)$ when X produces value $\langle v_1, \dots, v_n \rangle$.

If X has dimension n and ψ has type $n \rightarrow n'$, then $\psi(X)$ has dimension n' .

Observe that the types compose. If X has type $0 \rightarrow n$ and ψ has type $n \rightarrow n'$, then $\psi(X)$ has type $0 \rightarrow n'$.

The notation $\psi(X)$ is intended to evoke the physical transformation we are making on the FRP, passing the value produced by X through ψ . We are not literally evaluating the function ψ with argument X but rather co-opting the notation of evaluation to *operate* on the FRP using the function. Think of X in this expression as a “hole” that we will fill with X ’s value *when it becomes available*, passing that value to ψ . In the playground, you can use this notation `psi(X)` or the \wedge operator, `X \wedge psi`, as you prefer.

For simple functions like $\psi(x) = x^2$, $\varphi(x) = 4x - 3$, $\zeta(x_1, x_2) = -x_1x_2$, or $\xi(x) = e^{-2x}$, we often write the transformed FRPs with the statistic directly **inlined**, just writing the expression for the statistic in terms of the FRP itself, e.g., X^2 , $4X - 3$, $-X_1X_2$, or e^{-2X} . This is used for simple arithmetic, projections, and conditions¹⁹ and is convenient because we use such transformations so frequently.

Inlined Statistics. When working with a transform of an FRP X by a simple statistic, we often **inline** the statistic, writing the expression for the statistic in terms of X instead of an input parameter. This obviates the need to name the simple statistics we use.

We have already seen several examples of transformed FRPs. In Example 2.1, each roulette play is a statistic ψ of type $1 \rightarrow 1$ that takes a pocket and returns the monetary value of the play (negative for a loss). If R is the FRP representing the

¹⁹For projections, see subsection 2.3 below and Section F.7. For conditions and indicators, see Sections 5 and F.4. Anonymous functions in Section F.3 are also relevant.

pocket the ball lands in, then $\psi(R)$ represents the value of the play. Of course, once R is activated – that is, the ball lands – we know the value of the play, so our goal is to predict the value while R is still fresh, that is, before the spin.

In the previous section, we constructed the transformed FRP `DiceSum` that represents the sum of the values on three rolled dice. This is the transform of the 3-dimensional FRP `Roll`, representing the values of the three dice, by the statistic `Sum` of type $3 \rightarrow 1$. (`Sum` is built in to the playground, so we give it a name rather than using a generic variable like ψ or φ .) Here are a couple more examples.

Example 2.2. \mathcal{C} is a cube with side length 2, aligned with the coordinate axes, and centered on the origin. Let P be the FRP of dimension 3 whose value represents a random point from among the corners of \mathcal{C} , the midpoints of \mathcal{C} 's edges, the centers of \mathcal{C} 's faces, or the center of \mathcal{C} (the origin). How far is the point from the origin? For this question, we define the statistic φ of type $3 \rightarrow 1$ that computes the distance in space from its input to the origin:

$$\varphi(x, y, z) = \sqrt{x^2 + y^2 + z^2}.$$

The FRP $\varphi(P)$ has dimension 1 and size 4, with possible values $\sqrt{3}$, $\sqrt{2}$, 1, and 0. Its value answers our question.

Example 2.3. An Olympic archery target has 10 equal width rings (two in each of five colors: white, black, blue, red, gold). Any shot outside those rings counts as a miss, scoring 0 points. Otherwise, a shot is scored 1, 2, \dots , 10 by the region it hits, with 1 for the outermost white ring and increasing towards the center (gold). An archer shoots 6 arrows in a round.

Let A be the 6-dimensional FRP that represents the shots of one archer during a single round. The values of A are tuples whose entries are the archer's scores on the six successive shots. So, $\langle 2, 4, 10, 9, 7, 10 \rangle$ and $\langle 5, 6, 0, 1, 3, 10 \rangle$ are two possible values of A .

We define three relevant statistics of type $6 \rightarrow 1$.

- ψ is the archer's total score during the round;
- φ is the number of times the archer hits the target; and
- ζ is the number of times the archer hits the center-most “bull's eye” region

Here, we will define these statistics in two ways to make their meaning clearer – in code and mathematically. Look at both definitions for each function to see how they are doing the same thing.

In Python, we define these statistics like ordinary functions, except apply a decorator *decorator* (`@scalar_statistic`) before the definition so `frplib` knows they are statistics, which gives them useful attributes and convenient operations.

- ψ

```
@scalar_statistic(codim=6)
def psi(v)
    "Archer's total score in one round."
    return Sum(v)
```

- φ

```
@scalar_statistic(codim=6)
def phi(v)
    "Archer's number of hits in one round."
    hits = 0
    for shot in v:
        if shot > 0:
            hits += 1
    return hits
```

- ζ

```
@scalar_statistic(codim=6)
def zeta(v)
    "Archer's number of bull's eyes in one round."
    bulls_eyes = 0
    for shot in v:
        if shot == 10:
            bulls_eyes += 1
    return bulls_eyes
```

Mathematically, we use *indicators*, described in Section F.4, and summation notation $\sum_{i=1}^6 x_i = x_1 + \dots + x_6$. Our three statistics are defined as

- $\psi(v) = \sum_{i=1}^6 v_i$;
- $\varphi(v) = \sum_{i=1}^6 \{v_i > 0\}$, where the indicator $\{v_i > 0\}$ equals 1 if $v_i > 0$ or 0 otherwise; and
- $\zeta(v) = \sum_{i=1}^6 \{v_i = 10\}$, where the indicator $\{v_i = 10\}$ equals 1 if $v_i = 10$ or 0 otherwise.

A sum like $\sum_{i=1}^6 v_i$ is a mathematical analogue of a loop where we accumulate the sum one term at a time, giving $v_1 + v_2 + \dots + v_6$. A Boolean expression in Iverson braces, like $\{v_i > 0\}$, is a mathematical analogue of an if-then-else. *If* $v_i > 0$, then return 1 else return 0.

The transformed FRPs $\psi(A)$, $\varphi(A)$, and $\zeta(A)$ represent the total score, the number of hits, and the number of bull's eyes the archer shoots for the round. All three depend on the value of A and are activated as soon as A is, when the round is complete.

In a real round of archery, the archer takes one shot at a time, so at various points during the evolution of the random process we are observing, we have *partial information* about the data. We might want to use this partial information in practice. For instance, we might bet on the Archer's total score having observed the first three shots of the round.

This relates to the component FRPs in Figure 13. Here, the component FRPs are those representing the individual shots, call them A_1, A_2, \dots, A_6 . Each has dimension 1 and size 11, and A is a combination (specifically a *mixture* as described in Section 4) of these six FRPs. The A_i 's are activated at different times: A_1 after the first shot, A_2 after the second shot, and so on. And A is activated when all six have been activated.

We can construct an FRP, call it T , that represents the archer's first three shots. T is built from A_1, A_2, A_3 and activated when those three are activated. Consider a statistic ξ of type $3 \rightarrow 1$ that computes the total score from three shots. The transformed FRP $\xi(T)$ represents the archer's total score after three shots. This is a feature FRP in Figure 13 that depends on only some of the

component FRPs; the dashed arrow in the figure highlights the possibility of such dependence allowing some feature FRPs to be activated before all the data is available. This lets us use partial information while the random process is still evolving.

The two paths (dashed and undashed) in Figure 13 from Component FRPs to Feature FRPs give the same result. Here, $\xi(T) = \tilde{\xi}(A)$ with statistic $\tilde{\xi}$ of type $6 \rightarrow 1$ where $\tilde{\xi}(v) = \xi(v_1, v_2, v_3)$. $\tilde{\xi}(A)$ expresses this Feature FRP from the data (undashed path), and $\xi(T)$ directly from the components (dashed path).

2.2 Transformed Kinds

When we transform an FRP with a statistic, we get a new FRP whose kind is related to the Kind of the original FRP. Here, we define how to transform a *Kind* by a statistic. If K is a Kind and ψ is a statistic, we will denote the transformed Kind by $\psi(K)$, analogously to our notation for a transformed FRP.

It is essential that our definition of a transformed Kind be consistent with the definition of a transformed FRP in the sense of the following diagram:

$$\begin{array}{ccc} \text{FRPs} & \xrightarrow{\psi} & \text{FRPs} \\ \downarrow \text{kind} & & \downarrow \text{kind} \\ \text{Kinds} & \xrightarrow{\psi} & \text{Kinds} \end{array}$$

The nodes in this graph represent the set of FRPs and the set of Kinds; the edges represent operations that map one set into the other. The horizontal edges (labeled ψ) represent transformation by the statistic ψ ; the vertical edges (labeled kind) represent mapping an FRP to its Kind. Every path along the direction of the arrows represents a composition of these two operations. There are two paths in this graph from the top left to the bottom right. We can first transform an FRP by a statistic ψ then compute its kind, or we can compute the Kind of an FRP and then transform that Kind by the statistic ψ . *We require that both paths lead to the same result.*²⁰ In particular, if X is an FRP then

$$\text{kind}(\psi(X)) = \psi(\text{kind}(X)). \quad (2.1)$$

The Kind of the transformed FRP equals the transform of the original FRPs Kind.

²⁰See Section F.5. This requirement makes the graph a *commutative diagram*.

We can take equation (2.1) as the definition of a transformed Kind, yet it is also helpful to have a more operational definition. The key observation is that the possible values of a transformed FRP $\psi(X)$ are obtained from the possible values of X by applying ψ . So, we start by applying ψ to the value at every leaf node in $\text{kind}(X)$. However, it is possible that two distinct values of X map to the same value, so these two branches in $\text{kind}(X)$ would map to a single branch in $\text{kind}(\psi(X))$.

If two distinct values v and v' of X map to the same value $u = \psi(v) = \psi(v')$, then the weights of v and v' should add into the weight of u for the Kind of $\psi(X)$. To see why this is true, think about our experiments in Section 1.3. If we ran a demo with a large number of clones of X and transformed them by the statistic ψ , then every time either v or v' appears as a value of X , we get the value u for $\psi(X)$. So the proportions for v and v' add into the proportion for u in what we see. This gives us a procedure for computing the transformed Kind: apply ψ to the values at the leaf nodes of $\text{kind}(X)$ and then combine the branches that map to the same value, adding their weights.

Definition 4. If X is an FRP with Kind K and ψ is a compatible statistic, then the transformed Kind, denoted $\psi(K)$, equals the Kind of the transformed FRP $\psi(X)$, as in equation (2.1).

To find $\psi(K)$, we create a tree where

1. the leaf nodes have values in the set of $\psi(v)$ for v in the values of K ; and
2. the weight associated with a value u is the sum of weights in K for all values v with $u = \psi(v)$.

That is, **we first apply the statistic ψ to each value of K and then combine branches that map to the same value, adding their weights.**

Figures 14 and 15 show simple examples of this process, in two steps. First, we apply the statistic to the value at each leaf node, showing the new value for each leaf across the blue bar, which represents the action of the statistics adapter. Second, if these transformed values are equal for any leaf nodes, we combine their branches and add their weights, giving the transformed Kind shown on the right of each Figure. In the first Figure, several values map to each of $\langle 1 \rangle$ and $\langle 2 \rangle$, and these branches are combined. In the second Figure, each of the original values maps to a distinct value, so no branches are combined.

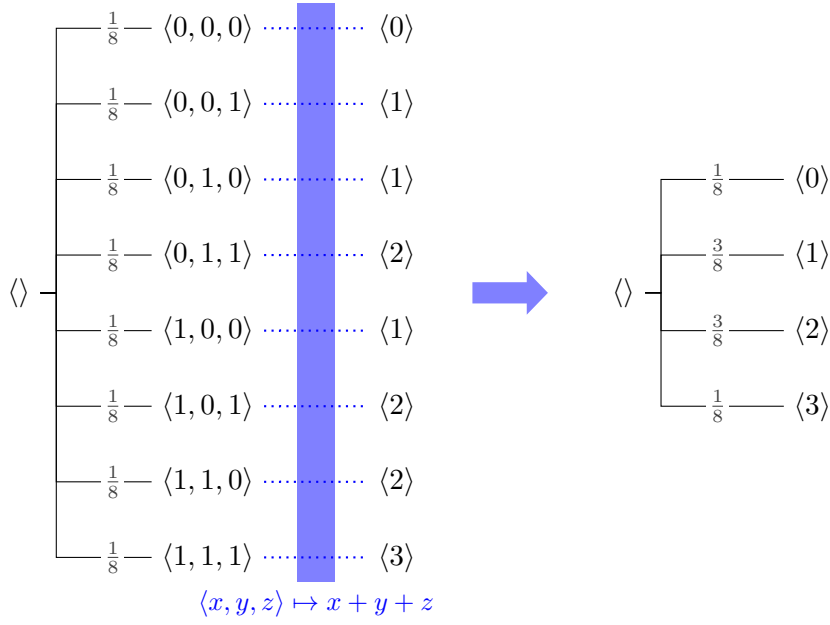


FIGURE 14. The transform of a three-dimensional Kind by the statistic that maps input $\langle x, y, z \rangle$ to $x + y + z$, shown in two steps. First, we apply the statistic to the values of the original Kind. Second, we combine branches that map to the same value, summing their weights to obtain the weight of the combined branch.

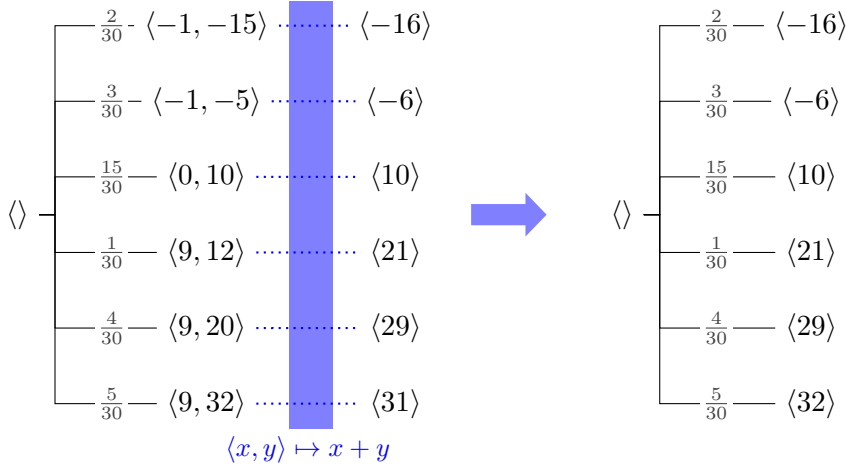


FIGURE 15. The transform of a two-dimensional Kind by the statistic that maps input $\langle x, y \rangle$ to $x + y$, shown in two steps. First, we apply the statistic to the values of the original Kind. Second, we combine branches that map to the same value, summing their weights, but here all values map to distinct values, so no branches are combined.

The Kinds in both these Figures have width 1. The same idea applies to any Kind: apply the statistic to the values and then combine branches that map to the same value. But if the Kind has width bigger than 1, we first convert it to “canonical form”, – which is always a width 1 tree – as described in Section 3 before transforming it.

Example 2.4. In Example 2.2, we considered an FRP that represents a random point chosen from the corners, edge midpoints, face centers, and center of a cube of edge-length 2 centered on the origin. We then defined the transform of this FRP by the statistic that computes the distance of a point to the origin. Assuming that the original FRP has a Kind with equal weights on every branch, we compute the transformed Kind as shown in Figure 16. This takes the two steps described earlier: apply the statistic to the values at every leaf node and then combine branches that map to the same value, adding their weights.

The result is consistent with our intuition. There are 8 corners, 12 edges, 6 faces, and 1 center, and the Kind shows that a large demo of the FRPs with the transformed Kind will give the distances from the origin to corner, edge midpoint, face center, and center in just these proportions.

Example 2.5. The Kind in Figure 7 describes FRPs that represent the roll of three, balanced, six-sided dice. Consider two questions: (i) What is the largest value among the dice? (ii) How many of each value did we see in the roll? We define statistics to capture these questions. Each will take a 3-tuple as input giving the values of the red, green, and blue dice. For (i), we want to compute the maximum of the three dice; this is a built-in statistic in the playground, called **Max**, so we just use that name here with type $3 \rightarrow 1$.

For each leaf node in Figure 7, we compute the maximum of the dice. Because there are only 6 possible maximums and 216 leaf nodes, multiple branches will have the same maximum. Combining those branches and adding their weights gives us the Kind:

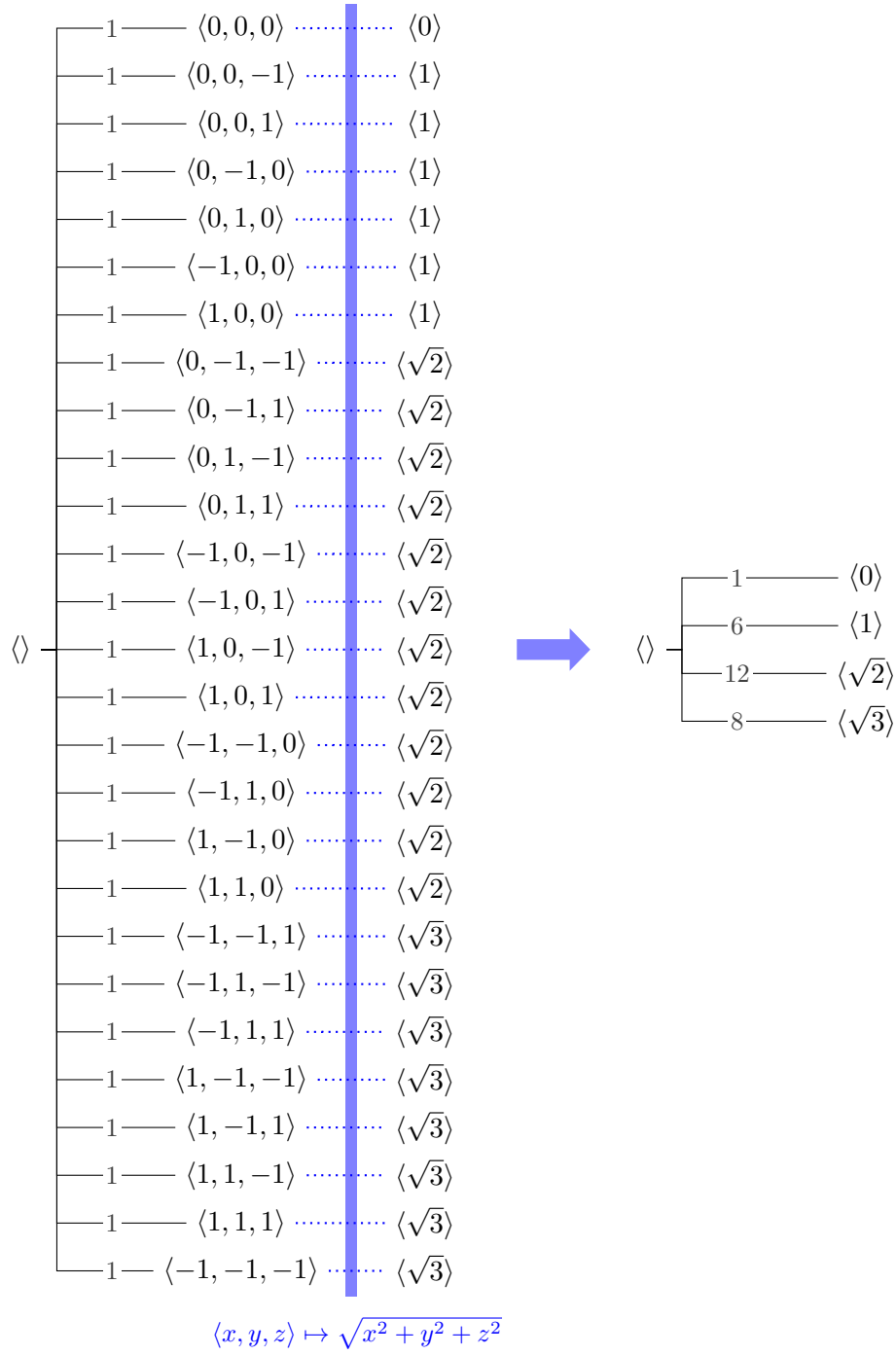
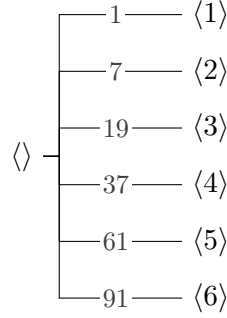


FIGURE 16. Computing the transformed Kind in Example 2.4.



There is one branch with maximum 1 (all three dice equal to 1), seven with maximum 2 (three with a 2 and two 1s, three with a 1 and two 2s, and one with three 2s), and so on up to 91 branches with a maximum of 6.

For question (ii), our statistic ξ will have type $3 \rightarrow 6$, returning a tuple v where v_i counts the number of dice with value i . Define ξ mathematically by

$$\begin{aligned} \xi(r, g, b) = & \langle \{r = 1\} + \{g = 1\} + \{b = 1\}, \{r = 2\} + \{g = 2\} + \{b = 2\}, \\ & \{r = 3\} + \{g = 3\} + \{b = 3\}, \{r = 4\} + \{g = 4\} + \{b = 4\}, \\ & \{r = 5\} + \{g = 5\} + \{b = 5\}, \{r = 6\} + \{g = 6\} + \{b = 6\} \rangle. \end{aligned}$$

Here, terms like $\{g = 3\}$ are *indicators*, as described in Section F.4. For instance, $\{g = 3\}$ is 1 when $g = 3$ and 0 otherwise. The term in the i th component of the tuple returned by ξ counts the number of dice equal to i .

We can define this statistic in the playground by defining this as a Python function. Again, we use a `@statistic` “decorator” to convert the function to a `Statistic` object with useful properties.

```
@statistic(dim=6)
def xi(r, g, b):
    "Counts the number of dice (r, g, and b) with each value."
    return [(r == i) + (g == i) + (b == i) for i in irange(1,6)]
```

We can pass `xi` individual values for the three dice or a single 3-tuple,²¹ and using the FRP `Roll` from earlier, we can compute the transformed FRP `xi(Roll)` and its Kind with *either* `kind(xi(Roll))` or `xi(kind(Roll))` by equation (2.1).

²¹Because `xi` has three explicit parameters, the playground can infer `codim=3` automatically. With a single parameter, it helps to give `codim` explicitly. If `xi` instead had parameters `r, g, b, *more`, it would have type $n \rightarrow 6$ for every $n \in [3..)$.

2.3 Projections and Marginals

If you have an FRP representing a random graph, you might focus on a subgraph. For an FRP representing a random image, you might have questions about one or more particular pixels. When modeling repeated roulette spins, you might analyze the outcome of bets on only some of those spins. This operation – *extracting* selected parts of a value – is so common and useful that it merits special attention. Statistics that extract one or more components from a tuple are called *projections*.

Definition 5. A **projection** is a statistic that maps a tuple to a tuple of smaller dimension containing only specified components of the original tuple.

A projection ψ of type $n \rightarrow m$, with $m \leq n$, has the form

$$\psi(\langle x_1, x_2, \dots, x_n \rangle) = \langle x_{i_1}, x_{i_2}, \dots, x_{i_m} \rangle \quad (2.2)$$

where $1 \leq i_1 < i_2 < \dots < i_m \leq n$ are the indices of the selected components.

Remember that we elide the distinction between lists of length 1 and scalars, so we can write a projection of type $n \rightarrow 1$ as $\psi(x) = x_i$ for $1 \leq i \leq n$.

Mathematically, we denote projection statistics specially: using the name **proj** with the selected indices in the subscript, as described in detail Section F.7. For instance,

$$\begin{aligned} \text{proj}_3(\langle 10, 20, 30, 40, 50 \rangle) &= 30 \\ \text{proj}_{3,5}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 30, 50 \rangle \\ \text{proj}_{1,3,5}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 30, 50 \rangle. \end{aligned}$$

If we want to project onto a *range* of components, say all the components from index i up to and including index j , we write the range as $i..j$ in the subscript. If either endpoint is missing, the range extends all the way to the corresponding end.

$$\begin{aligned} \text{proj}_{1..3}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 20, 30 \rangle \\ \text{proj}_{3..}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 30, 40, 50 \rangle \\ \text{proj}_{..3}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 20, 30 \rangle. \end{aligned}$$

We also use the functions with base name $\overline{\text{proj}}$ to indicate a projection that *excludes* the listed components. For instance,

$$\begin{aligned}\overline{\text{proj}}_3(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 20, 40, 50 \rangle \\ \overline{\text{proj}}_{3,5}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 20, 40 \rangle \\ \overline{\text{proj}}_{1,3,5}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 20, 40 \rangle \\ \overline{\text{proj}}_{1,2,3,5}(\langle 10, 20, 30, 40, 50 \rangle) &= 40 \\ \overline{\text{proj}}_{2..4}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 50 \rangle \\ \overline{\text{proj}}_{3..}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 20 \rangle \\ \overline{\text{proj}}_{..4}(\langle 10, 20, 30, 40, 50 \rangle) &= 50.\end{aligned}$$

In `frplib`, we access projections using the pre-defined statistics `Proj[indices...]`, where the indices start *counting from 1*. For example:

```
pgd> x = vec_tuple(10, 20, 30, 40, 50)
pgd> x
<10, 20, 30, 40, 50>
playground> Proj[3](x)
30
playground> Proj[3,5](x)
<30, 50>
playground> Proj[1,3,5](x)
<10, 30, 50>
```

(We can pass ordinary Python tuples to these statistics, but we use the function `vec_tuple` to obtain the more flexible tuples that are used for the values of FRPs and Kinds.) Between the brackets, `Proj` can accept multiple individual indices or a sequence of indices, like `Proj[(1,3,5)]` or `Proj[range(2,5)]`. Note that `range(a,b)` includes `a` but not `b`. These also accept Python “slices”, where `i : j` consists of indices from `i` up to but not including `j` (or to the end if `j` is excluded) and where `i : j : k` consists of indices from `i` up to but not including `j` skipping by `k`. So, `Proj[1:5:3]` is the same as `Proj[1,4]` and `Proj[1:6:2]` is the same as `Proj[1,3,5]`.

For convenience, we can use *negative indices* to indicate component indices *starting from the end*, with `-1` the last component, `-2` the second to last, and so forth.

```
pgd> Proj[-1](x)
50
pgd> Proj[-4,-3](x)
<20, 30>
pgd> Proj[2,-1](x)
<20, 50>
```

The family Projbar is also defined to mimic the $\overline{\text{proj}}$ functions:

```
pgd> Projbar[3](x)
<10,20,40,50>
pgd> Projbar[3,5](x)
<10,20,40>
pgd> Projbar[1,3,5](x)
<20, 40>
pgd> Projbar[1,2,3,5](x)
40
```

These also accept negative indices to count from the end.

As with any other statistic, we can use projections to transform FRPs and Kinds. But `frplib` also supports a shorthand using the `[]` operator with indices *directly* on the FRP or Kind. For example, if we make the following definitions

```
pgd> bits = uniform(0,1) ** 7
pgd> B = frp(bits)
pgd> first_bit = Proj[1]
pgd> odd_bits = Proj[1:8:2]
pgd> even_bits = Proj[2:8:2]
pgd> last_bit = Proj[-1]
pgd> last_two_bits = Proj[6:]
```

then we can apply the statistics in several equivalent ways

```
pgd> B
An FRP with value <0, 1, 1, 1, 1, 0, 1>
pgd> # Equivalent ways to apply first_bit (common value at end)
pgd> B ^ first_bit
```

```
pgd> first_bit(B)
pgd> B ^ Proj[1]
pgd> Proj[1](B)
pgd> B[1]
An FRP with value <0>
```

```
pgd> # Equivalent ways to apply odd_bits (common value at end)
pgd> B ^ odd_bits
pgd> odd_bits(B)
pgd> B ^ Proj[1:8:2]
pgd> Proj[1:8:2](B)
pgd> B[1:8:2]
An FRP with value <0, 1, 1, 1>
```

```
pgd> # Equivalent ways to apply last_bit (common value at end)
pgd> B ^ last_bit
pgd> last_bit(B)
pgd> Proj[-1](B)
pgd> Proj[7](B)
pgd> B[-1]
pgd> B[7]
An FRP with value <1>
```

Try these out yourself, and examine the results.

Puzzle 13. In the playground, demonstrate as many equivalent ways as you can to transform the FRP `B` by the statistics `even_bits` and `last_two_bits`.

The same approaches work with Kinds as well as FRPs. For instance, `bits` above is the Kind of `B`, and the following expressions are equivalent for producing the transformed Kind with `last_two_bits`:

```
pgd> bits ^ last_two_bits
pgd> last_two_bits(bits)
pgd> bits ^ Proj[6:]
```

```

pgd> Proj[6:](bits)
pgd> Proj[6,7](bits)
pgd> Proj[-2:](bits)
pgd> Proj[-2,-1](bits)
pgd> ProjBar[1:6](bits)
pgd> bits[6:]
pgd> bits[-2,-1]
pgd> bits[6,7]
      ,---- 1/4 ---- <0, 0>
      |---- 1/4 ---- <0, 1>
<> -|
      |---- 1/4 ---- <1, 0>
      ^---- 1/4 ---- <1, 1>

```

While this may initially seem like a lot of choices, there are just a few shorthands to keep in mind. The \wedge operator (think arrow) transforms an object (FRP, Kind, statistic) on the left side by a statistic on the right side. This is convenient to use when the statistic is derived from a factory or from a statistic expression (as described in the next subsection). A shorthand for this *looks like* a function call, where we pass the object to be transformed to the statistic. We tend to use this form when statistics have names because it matches our mathematical notation (e.g., $\psi(B)$) and captures the spirit of what the transform means. Because statistics are objects, they can be stored in variables and thus given names; when we use two expressions for the same statistic, the results are interchangeable. Thus for instance, `last_two_bits(bits)` and `Proj[6:](bits)` are the same because `Proj[6:]` is the statistic that we named `last_two_bits`. And finally for convenience, we allow direct application of the `[]` indexing operator to FRPs and Kinds, equivalent to the corresponding `Proj`, to make this common operation easier. Note that the indexing scheme – based on Python – provides multiple ways to generate the same set of indices, with lists or slices or counting from the end and so forth.

As Figure 13 suggests, we often build FRPs (and Kinds) that embody complicated information in high-dimensional tuples, so it is very common to work with transforms by projection. To express the relation between an FRP and another derived by projection and between a Kind and another derived by projection, we use the specific

label *marginal*.

Definition 6. If X is an FRP and $X' = \text{proj}_{i_1, i_2, \dots, i_m}(X)$, then X' is an FRP obtained by applying a projection statistic to X , then we call X' a **marginal (FRP)** of X . It is specifically identified by the indices i_1, i_2, \dots, i_m .

If k is a Kind and $k' = \text{proj}_{i_1, i_2, \dots, i_m}(k)$, then k' is Kind obtained by applying a projection statistic to k , then we call k' a **marginal (kind)** of k' . It is specifically identified by the indices i_1, i_2, \dots, i_m .

The process of transforming an FRP or Kind by a projection is sometimes called **marginalization**. The process of collecting the marginal FRPs for the projections onto every scalar component is called decomposing an FRP into its components.

Suppose X is an FRP of dimension n . We know that X produces a value that is a list of n numbers. The i^{th} component of X , for $i \in [1 \dots n]$,²² is just the FRP that gives us the i^{th} element of the list that X produces, which is exactly the value of the *transformed* FRP $\text{proj}_i(X)$. If we define $X_i = \text{proj}_i(X)$ for $i \in [1 \dots n]$, we call $\langle X_1, X_2, \dots, X_n \rangle$ the *components* of X .

²² $[1 \dots n]$ is an *increment*, the set of integers from 1 to n . See Section F.1.2.

Definition 7. If an FRP X has dimension n , then we can decompose it into **components**, *scalar* FRPs X_1, \dots, X_n with $X_i = \text{proj}_i(X)$ for each $i \in [1 \dots n]$. We call X_1, \dots, X_n the components FRPs of X , or just the components of X for short.

Puzzle 14. What are the components of the FRP B in the illustration on page 59? What are their Kinds?

Remember that, despite the special name, projections are just statistics, and thus satisfy all the properties of statistics. For instance, equation 2.1 holds, so if $X_i = \text{proj}_i(X)$, then $\text{kind}(X_i) = \text{proj}_i(\text{kind}(X))$. We get this kind by taking the i th component of each value and combining all the branches (adding their weights) that have the same value of this component.

2.4 Examples

In this section, we develop two extended examples that use statistics in interesting ways: to express and answer questions about the objects produced by a random process and as procedures for estimating the specification of a random system whose specification is unknown. These examples also illustrate how we use FRPs and Kinds to model random systems. Keep an eye out throughout for the patterns in Figure 13.

RANDOM GRAPHS. A **graph** is a mathematical structure that describes pairwise relationships among various entities. See Interlude F examples F.2.18 and F.2.19 for an overview and Interlude G for a detailed discussion. A graph has a set of nodes representing the entities and a set of edges representing the relationships between pairs of entities. Here, we will restrict our attention to *undirected, simple graphs with no loops*. This means that an edge connecting a pair of nodes has no preferred direction (undirected); that there can be at most one edge between any two nodes (simple); and that edges can only connect distinct nodes (no loops). Our nodes here are integers from 1 up to the total number of nodes, and so an edge can be specified by a *set* of two nodes, $\{i, j\}$, indicating that an edge connects nodes i and j in the graph.

We will construct FRPs that represent random generated graphs in this family and use statistics to interrogate and predict properties of these graphs. To load the tools we will need into the playground, enter

```
pgd> from frplib.examples.random_graphs import *
```

at the terminal prompt. The `*` loads all the available tools; you can also list specific functions to import if you prefer. To get details on what is available in the module, enter

```
pgd> import frplib.examples.random_graphs
pgd> help(frplib.examples.random_graphs)
```

Follow along with the computations as we proceed.


The function `random_graph` is an FRP *factory* that returns an FRP representing a random graph that meets our specification. In particular,²³ `random_graph(n, p)` returns a fresh FRP for a random graph on nodes $[1..n]$ for positive integer n and

²³These are called Erdős-Renyi random graphs.

$0 \leq p \leq 1$, with $p = 1/2$ the default if p is not supplied. The parameter p specifies the chance that any particular edge appears in the graph. With $p = 0$, the graph will have no edges; with $p = 1$, it will have every possible edge. A few examples:

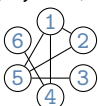
```
pgd> random_graph(3)
```

An FRP with value



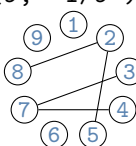
```
pgd> random_graph(6, '1/3')
```

An FRP with value



```
pgd> random_graph(9, '1/9')
```

An FRP with value



When you enter these into the playground, you will not see the pictures for these graphs. Instead, the FRPs returned by `random_graph` have values that represent the graph in a compact way: as tuples of 0s and 1s, where a 1 indicates that there is an edge between a particular pair of nodes.

The meaning of these tuples is illustrated in Figure 17 for several small values of n . The dimension of these tuples is $\frac{n(n-1)}{2} = \binom{n}{2}$, as this is the number of possible edges. The edges in a graph with n nodes can be described by an $n \times n$ “adjacency” matrix²⁴ A , where the entry in row i and column j , denoted A_i^j , equals 1 if there is an edge between nodes i and j and 0 otherwise. Because our graphs are undirected, the edge relation is symmetric, so $A_i^j = A_j^i$ for all i and j . Because our graphs have no loops, $A_i^i = 0$ for all i . As a consequence, the entire matrix A is determined by the upper right triangle, strictly above the main diagonal. For instance, in the matrix in the first row (and middle column) of Figure 17, the upper right triangle is highlighted, with $A_1^2 = 1$, $A_1^3 = 1$, and $A_2^3 = 0$. Reading these entries left-to-right and row-by-row as a single list gives the tuple $\langle 1, 1, 0 \rangle$, as shown in the right column in the Figure.

²⁴See sections F.9.2 and F.9.3 in Interlude F for more on matrices.

In the text, we will use the graph pictures wherever possible in lieu of the tuples, for clarity, but you will see the tuples when you play with these in the playground. The `show_graph` function takes a value tuple from a random graph or a random graph FRP and pops up a picture of the graph in a browser tab. The `adjacency_matrix`

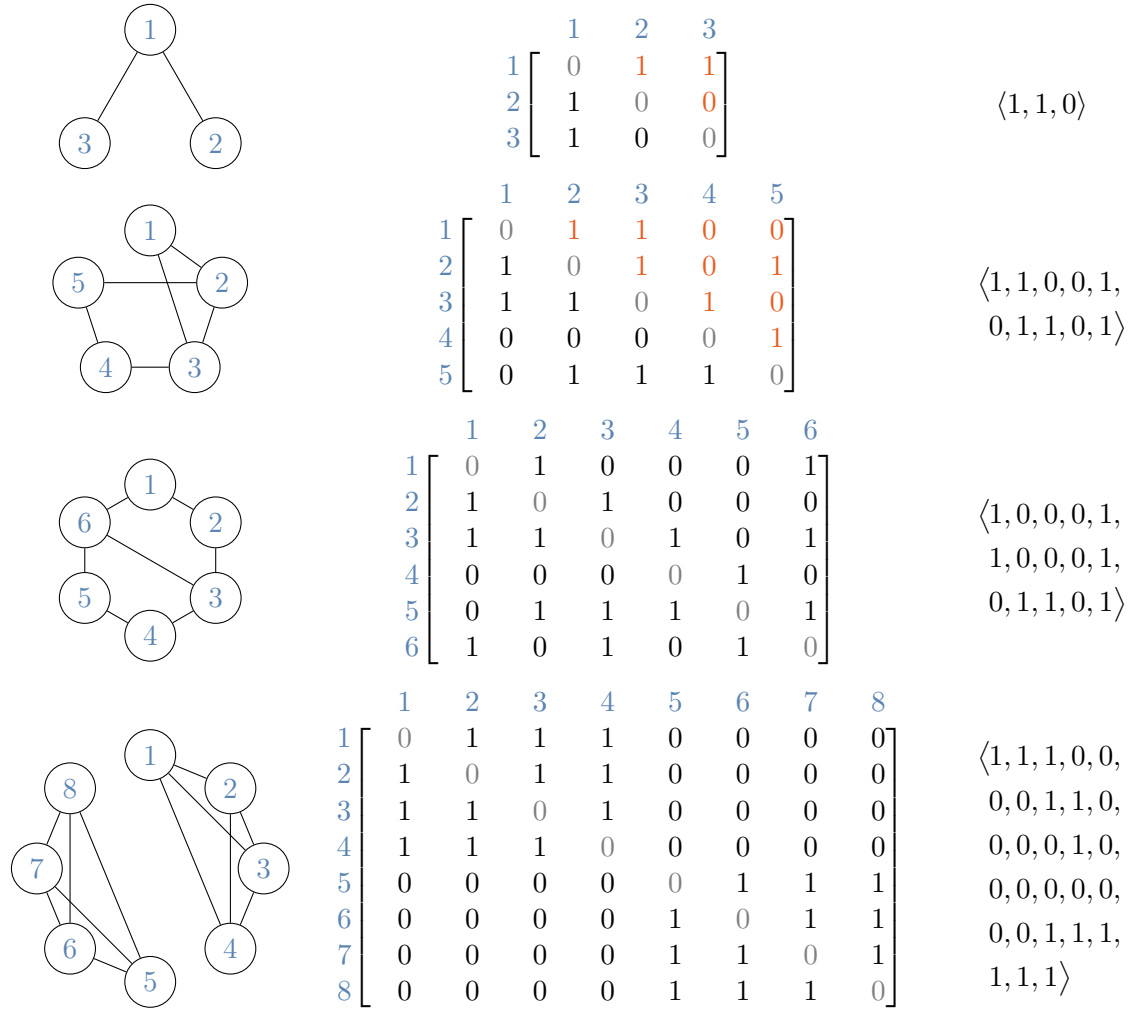


FIGURE 17. The representation of undirected simple graphs without loops as numeric tuples. The left column shows the graph. The middle columns gives the graph's "adjacency" matrix that shows which pair of nodes have an edge between them (with a 1 in the corresponding entries). The tuple is laid row-wise into the upper right triangle of the matrix, strictly above the main diagonal, and reflected by symmetry column-wise in the lower left triangle. The main diagonal entries are always zero because there are no loops. The right column gives the corresponding tuple representation, a list of 0s and 1s.


and `adjacency_list` functions takes a tuple or FRP and show you the edges in a different way, as illustrated below.

We start with $n = 3$ nodes to get a feel for the possibilities.

```
pgd> G_0 = random_graph(3)
```

```
pgd> G_0
```

An FRP with value



```
pgd> show_graph(G_0)
```

An FRP with value <1, 0, 1> # Picture shows in web browser not in terminal

```
pgd> adjacency_matrix(G_0)
```

```
  1 2 3
1 0 1 0
2 1 0 1
3 0 1 0
```

```
pgd> adjacency_list(G_0)
```


```
{1: [2], 2: [1, 3], 3: [2]}
```

For each node, the adjacency list gives the nodes connected to it by an edge. Note that each call to `random_graph` returns a fresh clone, so `G_0` is not the same as the earlier FRP with $n = 3$. Similarly,

```
pgd> G_1 = random_graph(3)
```

```
pgd> G_1
```

An FRP with value



```
pgd> adjacency_matrix(G_1)
```

```
  1 2 3
1 0 0 0
2 0 0 1
3 0 1 0
```

```
pgd> adjacency_list(G_1)
```

```
{2: [3], 3: [2]}
```

Now we can ask some questions of our random graphs. We are interested in various properties of the graphs produced by these FRPs. For instance: Are two

particular nodes connected by an edge? How many edges are associated with each node? How many total edges are in the graph? Can any node be reached from any other by following edges? If not, how many “connected components” does the graph have? Are there cyclical paths? Is the graph a tree? For each such question, we formulate a statistic that answers that question when given a graph on input. Our emphasis is not on answering the questions for a particular graph, but rather on *predicting the answer* for any setting of n and p . In what follows, we will exemplify this for a variety of statistics and graphs, and along the way, we will have to face some of the computational and mathematical challenges in answering our questions.

The function `has_edge` is a *statistic factory* because it *creates a statistic* that tests whether a graph has an edge between two specified nodes. For instance, `has_edge(2, 4)` is a statistic that tests whether there is an edge between nodes 2 and 4. We view the statistic’s return value as a Boolean, with 0 for false (\perp) and 1 for true (\top). A Boolean statistic is called a *condition* because indicates when a particular condition is met. We can check whether `G_0` and `G_1` have various edges:

```
pgd> G_0 ^ has_edge(1,3)
<0>
pgd> G_1 ^ has_edge(2,3)
<1>
pgd> has_edge(1,2)(G_0)
<1>
pgd> has_edge(1,2)(G_1)
<0>
pgd> edge13 = has_edge(1,3)
pgd> edge13(G_1)
<0>
```

Here, we see three equivalent ways of using the statistic returned from the factory to transform an FRP: using the `^` operator, applying the statistic directly, and naming the statistic to apply it later by name. The `^` operator makes the meaning of the operation clearer in this case, so it is particularly useful when transforming with statistic factories. An FRP with (Boolean) values 0 and 1 – as produced by transforming another FRP with a condition – is called an **event**. If the FRP has value 1, the event is said to have *occurred*, otherwise not.

The statistics `is_connected`, `is_acyclic`, and `is_tree` are conditions that test, respectively, (i) if every node can be reached from every other node by following edges, (ii) if the graph has no cycles,²⁵ and (iii) if the graph is a *tree* – a connected, acyclic graph.

²⁵A cycle exists if there is a non-trivial path along distinct edges from some node to itself.

```
pgd> is_connected(G_0)
An FRP with value <1>
pgd> is_connected(G_1)
An FRP with value <0>
pgd> is_acyclic(G_0)
An FRP with value <1>
pgd> is_tree(G_1)
An FRP with value <0>
```

The statistic `connected_components` has type $\binom{n}{2} \rightarrow n$. It maps a graph's tuple representation to a tuple with one integer component per node, and two nodes have the same integer if they are connected by a path in the graph. For example:

```
pgd> connected_components(G_0)
An FRP with value <1, 1, 1>
pgd> connected_components(G_1)
An FRP with value <1, 2, 2>
```

We will see other graph statistics below.

The Kinds for `random_graph(3,p)` are shown in Figure 18 for $p = 1/2$, $p = 1/3$, and $p = 3/5$. These have dimension 3 and size $8 = 2^3$. If you compute these Kinds in the playground, it will show weights with the same relative size but normalized so that they sum to 1. As we saw in subsection 1.3, Kinds whose weights differ by a constant scaling factor will produce the same demos and so are in practice equivalent.

When $p > 1/2$, the weights increase with the number of edges in the graph, as in Kind on the right of Figure 18. When $p < 1/2$, the weights decrease with the number of edges, as in Kind on the middle of Figure 18. To understand this, let us look at the Kind of the event that a particular edge (say $\{2, 3\}$) is in the graph. We can compute the Kind of the random graph FRP transformed by `has_edge(2,3)` or transform the Kind of the random graph FRP by `has_edge(2,3)`. By equation (2.1), both give the same result.

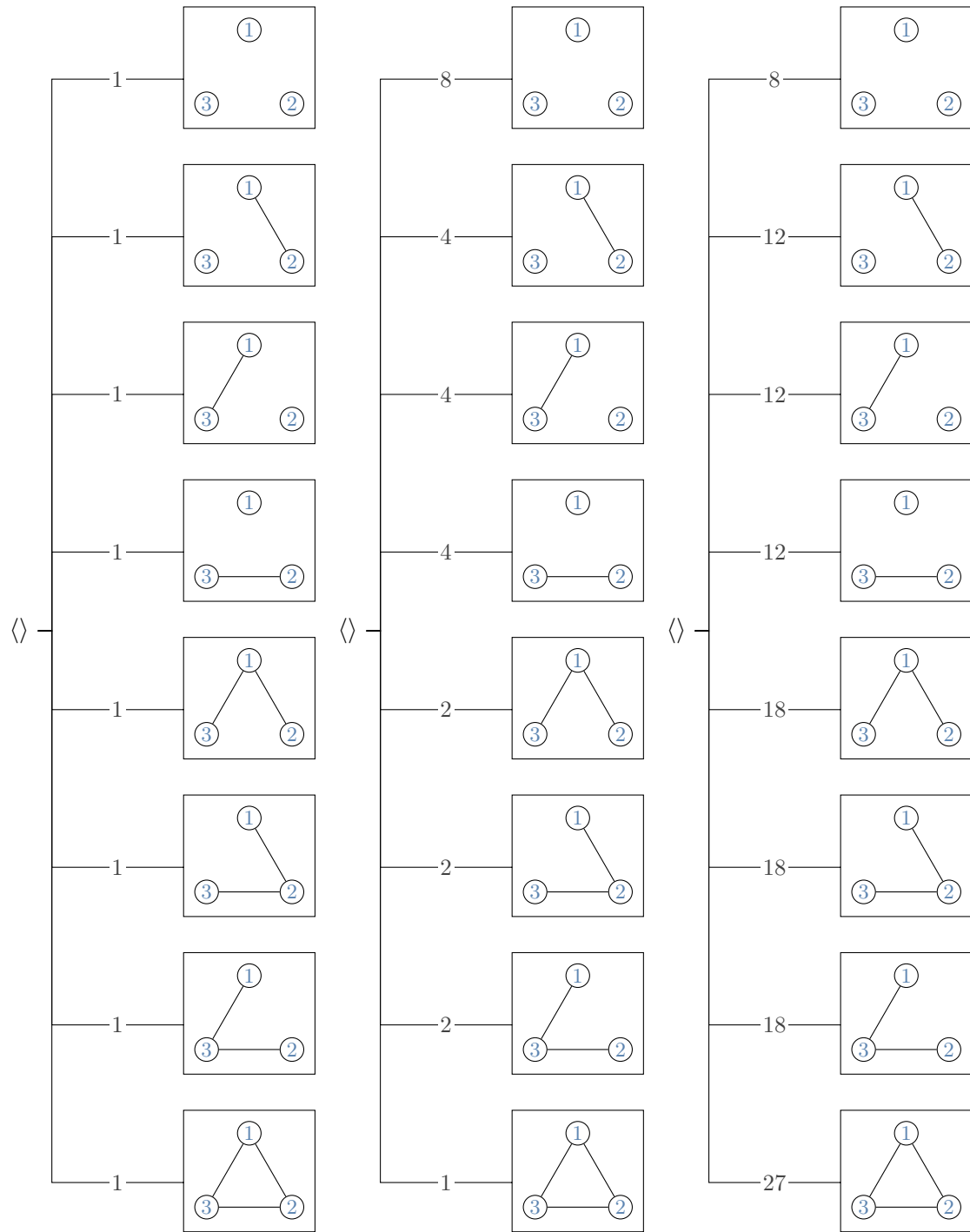


FIGURE 18. The Kinds of $\text{random_graph}(3)$ (left), $\text{random_graph}(3, 1/3)$ (center), and $\text{random_graph}(3, 3/5)$ (right). The Kinds are not in canonical form.

```

pgd> kind(random_graph(3,1/2) ^ has_edge(2,3))
      ,---- 1/2 ---- 0
<> -|
      `---- 1/2 ---- 1
pgd> kind(random_graph(3,1/3)) ^ has_edge(2,3)
      ,---- 2/3 ---- 0
<> -|
      `---- 1/3 ---- 1
pgd> kind(random_graph(3,3/5)) ^ has_edge(2,3)
      ,---- 2/5 ---- 0
<> -|
      `---- 3/5 ---- 1

```

Recall the steps for transforming a Kind: apply the statistic to the value at every leaf node then combine branches that map to a common value by adding the weights. Referring to Figure 18, we see that the first three and the fifth branches have values without a $\{2,3\}$ edge, and the others have values with a $\{2,3\}$ edge. Summing the weights for these branches: when $p = 1/2$, we get 4 without the edge and 4 with; when $p = 1/3$, we get 18 without and 9 with; when $p = 3/5$, we get 50 without and 75 with. This gives respective Kinds

$$\begin{array}{ccc}
\langle \rangle \begin{array}{l} \text{---} 4 \text{---} \langle 0 \rangle \\ \text{---} 4 \text{---} \langle 1 \rangle \end{array} & \langle \rangle \begin{array}{l} \text{---} 18 \text{---} \langle 0 \rangle \\ \text{---} 9 \text{---} \langle 1 \rangle \end{array} & \langle \rangle \begin{array}{l} \text{---} 50 \text{---} \langle 0 \rangle \\ \text{---} 75 \text{---} \langle 1 \rangle \end{array}
\end{array}$$

where the weights on 0 and 1 have ratios 1 to 1, 2 to 1, and 2 to 3. Renormalizing the weights to sum to 1, we get the same Kind the playground computes.

It seems that for these `random_graph(3,p)` Kinds, the ratio of the weight on $\langle 1 \rangle$ to the weight on $\langle 0 \rangle$ is $p/(1-p)$. Let's think about what this means in terms of the demos we ran in subsection 1.3. If we obtain a large number of `random_graph(3,p)` FRPs, check if they have a $\{2,3\}$ edge (i.e., transform them with `has_edge(2,3)`), and tabulate the results, we will see approximately *a proportion p of them have such an edge* and a proportion $1-p$ will not.

Puzzle 15. Do this test yourself for a variety of $0 \leq p \leq 1$. Use

```
FRP.sample(random_graph(3,p) ~ has_edge(2,3), 1_000_000)
```

to compute a demo for each p you choose.

Are the results consistent with the Kind? Do the results depend on which edge you choose?

The vast, apparently infinite number of FRPs stored at the Warehouse are not arranged in any systematic way; when you obtain one from the market, it is an arbitrary clone of the FRPs of its Kind. The demo tells us that if we take all the FRPs whose Kind is `kind(random_graph(3,p))`, activate them, and apply `has_edge(2,3)` to their value, the *average* of the results will be p . So p is the proportion of `random_graph(3,p)` FRPs for which the event that it has an edge $\{2,3\}$ occurs. In that sense, p quantifies the chance that you receive an FRP that represents a graph with such an edge. This is one way to think about the idea of *probability*: the probability of an event is the average value of the condition, a proportion, over all FRPs in the Warehouse of the same Kind.

An important observation for the future:

```
pgd> E(random_graph(3,1/2) ~ has_edge(2,3))
1/2
pgd> E(random_graph(3,1/3) ~ has_edge(2,3))
1/3
pgd> E(random_graph(3,3/5) ~ has_edge(2,3))
3/5
```

The expectation – the risk-neutral price, our prediction of the FRP’s value – is exactly that average! The probability that an event occurs is just our best prediction of whether its defining condition is true.

Puzzle 16. In the playground, compute the transformed Kind of `random_graph(3,p)` by the statistic `is_connected`, for p equal to $1/2$, $1/3$, and $3/5$. What is the probability that you obtain a connected graph in these three cases?

Answers: $5/18$, $2/3$, $7/10$

You might want to return to this after reading Section 5.

Aside. We might ask another type of question about the presence of edges: if we have a `random_graph(3,p)` FRP X and we have observed that the value of the transformed FRP $X \sim \text{has_edge}(1,2)$ is 1 – that is, we *know* that X ’s value has an edge $\{1,2\}$ – what can we say about whether X has an edge $\{2,3\}$? To incorporate our partial knowledge of X ’s value, we use a *conditional constraint* as discussed in detail in Section 5. For instance,

```
pgd> X = random_graph(3, '1/3')
pgd> kind(X | has_edge(1,2)) ^ has_edge(2,3)
      ,---- 2/3 ---- 0
<> -|
      `---- 1/3 ---- 1
```

The `|` is read as “given”; it takes an FRP or Kind on the left and a condition on the right and applies the constraint that the condition is true. We see here that the partial information about edge $\{1,2\}$ does not change the Kind of $X \sim \text{has_edge}(2,3)$. We will explore this more later.

As will become clearer in Section 4, for each p , the events

$$\text{random_graph}(3,p) \sim \text{has_edge}(i,j)$$

over all edges $\{i,j\}$ are the *component FRPs* of Figure 13 from which `random_graph(3,p)` is built.

In particular, for any p of your choosing such as $3/5$, try

```
pgd> edge_kind = weighted_as(0, 1, weights=[1 - p, p])
pgd> K_p = edge_kind * edge_kind * edge_kind
pgd> Kind.equal(K_p, kind(random_graph(3,p)))
True
pgd> frp(edge_kind) * frp(edge_kind) * frp(edge_kind)
An FRP with value <1, 0, 1>
```

Here, the function `weighted_as` is a *Kind factory*; look at `edge_kind` and see that it is Kind of the `has_edge(i,j)` transformed FRP that we saw above. The independent mixture (with operator `*`) builds a three-dimensional Kind that equals our random graph Kind. The last line builds the corresponding FRP, whose value is of the right type.

We can ask other questions as well and define statistics to represent them. How many edges does the graph have? (`edge_count`) How many connected components does the graph have? (`connected_component_count`) What are they? (`connected_components`) How many neighbors does each node have? (`degrees`) Is the graph free of cycles? (`is_acyclic`) It is useful to compute these first for random graphs with 3 nodes because the Kinds have small enough size that we can easily see how the transformation steps are working.

```
pgd> kind(random_graph(3,1/2)) ^ edge_count
      ,---- 1/8 ---- 0
      |---- 3/8 ---- 1
<> -|
      |---- 3/8 ---- 2
      `---- 1/8 ---- 3
pgd> kind(random_graph(3,1/3)) ^ edge_count
      ,---- 8/27 ----- 0
      |---- 12/27 ----- 1
<> -|
      |---- 6/27 ----- 2
      `---- 1/27 ----- 3
pgd> kind(random_graph(3,3/5)) ^ edge_count
      ,---- 8/125 ---- 0
      |---- 36/125 --- 1
<> -|
      |---- 54/125 --- 2
      `---- 27/125 --- 3
```

Calculate these Kinds by hand from Figure 18 and compare your results.

```
pgd> K_G3h = kind(random_graph(3))
```

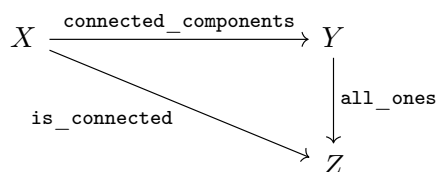
```
pgd> connected_components(K_G3h)
,---- 4/8 ---- <1, 1, 1>
|---- 1/8 ---- <1, 1, 2>
<> -+---- 1/8 ---- <1, 2, 1>
|---- 1/8 ---- <1, 2, 2>
`---- 1/8 ---- <1, 2, 3>
```

This Kind describes the connected components of `random_graph(3)`'s; the values give a label for each node (with labels starting from 1) where two nodes with the same label belong to the same connected component. Observe that from this information we can also answer other questions. Is the graph connected? It is if all components' labels are equal to 1. How many connected components are there? It is the largest component label. What is the *pattern* of component sizes? List the sizes of the components in decreasing order. We can answer these questions in two ways, either directly from the Kind `K_G3h` or by further transforming `connected_components(K_G3h)`. For instance,

```
pgd> all_ones = condition(lambda v: all(vi == 1 for vi in v))
pgd> K_G3h ^ connected_components ^ all_ones
,---- 1/2 ---- 0
<> -|
`---- 1/2 ---- 1
pgd> is_connected(K_G3h)
,---- 1/2 ---- 0
<> -|
`---- 1/2 ---- 1
```

The statistic `all_ones` is a condition that returns true when all components of its input equal 1. (The `lambda` syntax in Python defines *anonymous functions*, which are discussed in Section F.3. The term `lambda args: expr` is a function taking `args` as arguments and returning the value of expression `expr`, which may involve the given arguments.) Here, we use the ability of the `^` operator to *chain* several statistics together,²⁶ transforming first by `connected_components` then by `all_ones`. The statistic `is_connected` is *equal* to the statistic `connected_components ^ all_ones` that first applies `connected_components` and then applies `all_ones` to the returned result. We can visualize these relationships in a diagram

²⁶See Section F.5 for a detailed discussion of function composition.



where X , Y , and Z all stand for Kinds or for FRPs and the arrows represent transformations by statistics, with any two distinct paths from one node to another representing equal functions.

Similarly, using the built-in statistic `Max` to compute the maximum label:

```
pgd> K_G3h ^ connected_components ^ Max
      ,---- 4/8 ---- 1
<> -+---- 3/8 ---- 2
      `---- 1/8 ---- 3
pgd> connected_component_count(K_G3h)
      ,---- 4/8 ---- 1
<> -+---- 3/8 ---- 2
      `---- 1/8 ---- 3
```

And

```
pgd> K_G3h ^ connected_components ^ connected_component_sizes
      ,---- 1/8 ---- <1, 1, 1>
<> -+---- 3/8 ---- <2, 1, 0>
      `---- 4/8 ---- <3, 0, 0>
```

where the statistic `connected_component_sizes` takes a tuple of component labels and returns the sizes of the components in descending order. The 0's are padded out to n entries to ensure that our statistic has a fixed dimension as required. Thus the `<2, 1, 0>` value occurs in the three labelings of components where there is one component of size 2 and one component of size 1. Again, it is worth deriving these transformed Kinds by hand where the sizes are small enough to understand the operation clearly.

Another common question we might ask about a random graph is how many neighbors each node has. We can use the statistic `degrees` to answer this question.

```
pgd> degrees(K_G3h)
      ,---- 1/8 ---- <0, 0, 0>
```

```

|---- 1/8 ---- <0, 1, 1>
|---- 1/8 ---- <1, 0, 1>
|---- 1/8 ---- <1, 1, 0>
<> -|
|---- 1/8 ---- <1, 1, 2>
|---- 1/8 ---- <1, 2, 1>
|---- 1/8 ---- <2, 1, 1>
`---- 1/8 ---- <2, 2, 2>

```

But more typically we are less interested in what happens at particular nodes than at understanding the *pattern* in the numbers of neighbors globally over the graph. For this we can transform the previous Kind by the statistic **Ascending** that sorts the tuple in increasing order:²⁷

²⁷There is also an analogous **Descending** statistic.

```

pgd> K_G3h ^ degrees ^ Ascending
,---- 1/8 ---- <0, 0, 0>
|---- 3/8 ---- <0, 1, 1>
<> -|
|---- 3/8 ---- <1, 1, 2>
`---- 1/8 ---- <2, 2, 2>

```

This elides the distinction between degree profiles that differ only by labeling of the nodes. Observe how such profiles are combined into a single pattern and their weights added by the Kind transformation.

Puzzle 17. Explain the following computations with reference to Figure 18.

```

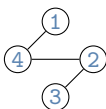
pgd> is_acyclic(K_G3h)
,---- 1/8 ---- 0
<> -|
`---- 7/8 ---- 1
pgd> kind(random_graph(3, 3/5)) ^ is_acyclic
,---- 27/125 ---- 0
<> -|
`---- 98/125 ---- 1

```

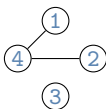
For $n = 4$ nodes, `random_graph(4,p)` has dimension 6 and size $64 = 2^6$.

```
pgd> G = random_graph(4)
```

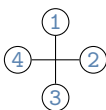
```
pgd> G
```

An FRP with value 

```
pgd> clone(G)
```

An FRP with value 

```
pgd> clone(G)
```

An FRP with value 

Look at `kind(random_graph(4))`, `kind(random_graph(4, 1/3))`, and `kind(random_graph(4, 3/5))` in the playground.

We can now ask some questions about random graphs with 4 nodes. equal to `kind(random_graph(4))`:

```
pgd> K_G = kind(G)    # equal to kind(random_graph(4))
```

How many edges does the graph have?

```
pgd> edge_count(K_G)
,---- 1/16 ---- 0
|---- 4/16 ---- 1
<> -+---- 6/16 ---- 2
    |---- 4/16 ---- 3
    `---- 1/16 ---- 4
```

Try answering this same question with `random_graph(4,1/3)` and `random_graph(4,3/5)` and compare the resulting Kinds.

How many connected components does the graph have?

```
pgd> connected_component_count(K_G)
,---- 38/64 ----- 1
|---- 19/64 ----- 2
<> -|
```

```
|---- 6/64 ----- 3
`---- 1/64 ----- 4
```

Notice how the weights shift towards fewer components when p is larger and towards more components when p is smaller.

```
pgd> kind(random_graph(4, '3/5')) ^ connected_component_count
,---- 0.76550 ----- 1
|---- 0.19354 ----- 2
<> -|
|---- 0.036864 ---- 3
`---- 0.004096 ---- 4
pgd> kind(random_graph(4, '1/3')) ^ connected_component_count
,---- 0.27572 ----- 1
|---- 0.37311 ----- 2
<> -|
|---- 0.26337 ----- 3
`---- 0.087791 ---- 4
pgd> kind(random_graph(4, '1/20')) ^ connected_component_count
,---- 0.0018012 ---- 1
|---- 0.030973 ----- 2
<> -|
|---- 0.23213 ----- 3
`---- 0.73509 ----- 4
```

Is the graph a tree? Is it connected? Is it free of cycles?

```
pgd> is_tree(G)
An FRP with value <1>
pgd> is_tree(K_G)
,---- 11/16 ---- 0
<> -|
`---- 5/16 ----- 1
pgd> is_connected(G)
An FRP with value <1>
pgd> is_connected(K_G)
```



```

      ,---- 13/32 ---- 0
<> -|
      `---- 19/32 ---- 1
pgd> is_acyclic(G)
An FRP with value <1>
pgd> is_acyclic(K_G)
      ,---- 13/32 ---- 0
<> -|
      `---- 19/32 ---- 1

```

For small p , we saw that the graph is likely to have many connected components and

```

pgd> kind(random_graph(4, '1/20')) ^ is_acyclic
      ,---- 0.00051509 ---- 0
<> -|
      `---- 0.99948 ----- 1

```

suggesting that the graph is very likely to be a *forest*²⁸ of small trees.

²⁸A *forest* is a collection of trees.

Does the graph have edges $\{1,2\}$ and $\{3,4\}$? For this question, we create a statistic using a *combinator*: a function that takes two or more statistics and returns a new one. In this case we use the **And** combinator.

```

pgd> has_12_34 = And(has_edge(1,2), has_edge(3,4))
pgd> has_12_34(G)
An FRP with value 0
pgd> has_12_34(K_G)
      ,---- 3/4 ---- 0
<> -|
      `---- 1/4 ---- 1

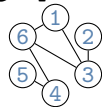
```

In all these cases, transforming the FRP with the statistic gives us an answer to the corresponding question *for that graph*, and transforming the Kind tells us how the answer varies over all FRPs of the original Kind. The transformed Kind lets us *predict* the answer to the question.

Generating random graphs is a relatively fast operation, so we can generate random graphs of substantial size and interrogate their properties.

```
pgd> G6 = random_graph(6)
```

An FRP with value



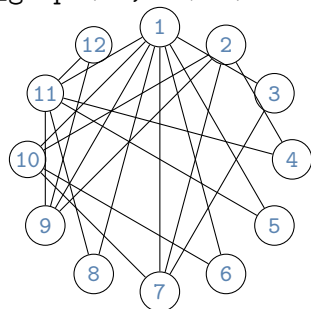
```
pgd> G6 ^ Fork(is_connected, is_acyclic, is_tree)
```

An FRP with value <1, 0, 0>

Here, `Fork` is a statistic combinator that applies the listed statistics in parallel to the same values and concatenates the results in order into one tuple. So we see that `G6` is connected because the first component of the tuple is 1 and that it has cycles and is not a tree because the last two components are 0.

```
pgd> G12 = random_graph(12, '1/3')
```

An FRP with value



```
pgd> is_tree(G12)
```

An FRP with value <0>

```
pgd> connected_components(G12)
```

An FRP with value <1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1>.

(It may be slow to evaluate its kind.)

```
pgd> G12 ^ connected_components ^ connected_component_sizes
```

An FRP with value <12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0>.

(It may be slow to evaluate its kind.)

```
pgd> G12 ^ degrees ^ ascending
```

An FRP with value <2, 2, 2, 2, 2, 2, 4, 4, 4, 4, 6, 8>.

(It may be slow to evaluate its kind.)

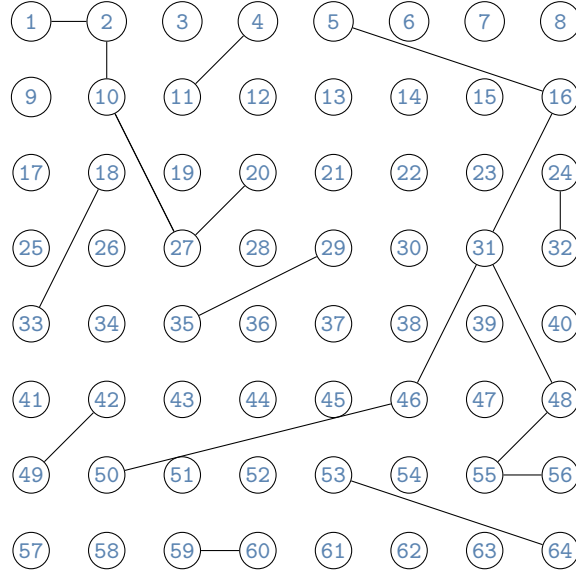
```
pgd> G12 ^ Fork(edge_count, degrees ^ Sum / 2)
```

An FRP with value <21, 21>.

Even with only 21 out of 66 possible edges, `G12` is connected.

```
pgd> G64 = random_graph(64, '1/128')
```

An FRP with value



```
pgd> is_forest(G64)
```

An FRP with value <1>

The transformed FRPs here describe the properties of the graphs represented by G6 and G12. Though we can generate FRPs easily, the playground is warning us that computing the Kind of G12, which has size 2^{66} , might be ... time consuming. Because p is small for G64, most of the nodes of this graph are isolated, leaving a forest of small trees. The condition `is_forest` tests whether a graph's connected components are all trees. It is defined using the function `ForEachComponent`, from `frplib.examples.random_graphs`, that takes a graph statistic (here `is_tree`) and applies it to the subgraph of each connected component.

As discussed earlier, our main goal is not just to interrogate the properties of particular graphs but to predict the properties of the random graphs before we activate the FRPs. Our basic approach to doing this is to compute the transformed *Kinds* with the statistics that represent our questions. When $n = 6$, the Kind of `random_graph(n,p)` has size $2^{15} = 32,768$. We can compute its Kind directly, but we can see that the size is growing fast. Indeed, for `random_graph(8,p)` the Kind has size $2^{28} = 268,435,456$, which is within the range of computational feasibility to compute but rather inconvenient. For general n , we get $2^{n(n-1)/2}$ – exponential

growth! Directly computing the Kinds is not always computationally feasible, so we need to bring other strategies to bear.

First, if we can compute directly with reasonable efficiency, we do, and for many problems this is enough. When that gets difficult – such as when the Kind of the data FRP has a large size – we can focus our attention on computing the Kinds of the feature FRPs (transforms of the data FRP relevant to our questions, cf. Figure 13), which usually have much smaller size and dimension. We can try to devise computational methods and algorithms that allow us to efficiently compute answers to our questions. And where possible and necessary, we apply *mathematical reasoning* to derive those answers, or approximate those answers to specified accuracy if an exact answer is elusive, or if necessary, put useful bounds on the answers. Ideally, we could answer our questions in full generality, but often we face a trade-off between the precision of our answers and the specificity of the assumptions we need to make. As such we often tailor our reasoning and analysis to specific questions or special cases that are amenable to analysis. The combination and interaction of computational and mathematical reasoning offers us a diverse and powerful range of tools for answering probabilistic questions. We will see this theme reprised many times as we proceed.

As an example, suppose we want to predict the number of edges in a graph like **G64**. The Kind of the FRP **G64** has size 2^{2015} , so direct computation is quite out of reach. We want the Kind of the transformed FRP, $\text{kind}(\mathbf{G64}) \sim \text{edge_count}$, but computing it *that way* is consequently infeasible. However, using a computational technique we will describe in Section 6.1, we can find the Kind quickly to use in practice. The function `fast_edge_count` in `frplib.examples.random_graphs` computes the Kind of `random_graph(n,p) \sim \text{edge_count}` for a specified n and p . For instance:

```
pgd> E64 = fast_edge_count(64, '1/128')
pgd> is_kind(E64)
True
pgd> clean(E64, '1e-6')
,---- 1.3583E-7 ----- 0
|---- 0.0000021562 ---- 1
|---- 0.000017105 ----- 2
|---- 0.000090420 ----- 3
|---- 0.00035830 ----- 4
```

----	0.0011353	-----	5
----	0.0029961	-----	6
----	0.0067741	-----	7
----	0.013395	-----	8
----	0.023532	-----	9
----	0.037188	-----	10
----	0.053399	-----	11
----	0.070253	-----	12
----	0.085273	-----	13
----	0.096064	-----	14
----	0.10096	-----	15
----	0.099416	-----	16
----	0.092094	-----	17
----	0.080532	-----	18
----	0.066682	-----	19
<> -+----	0.052427	-----	20
----	0.039236	-----	21
----	0.028016	-----	22
----	0.019125	-----	23
----	0.012505	-----	24
----	0.0078458	-----	25
----	0.0047308	-----	26
----	0.0027455	-----	27
----	0.0015356	-----	28
----	0.00082890	-----	29
----	0.00043229	-----	30
----	0.00021807	-----	31
----	0.00010651	-----	32
----	0.000050422	-----	33
----	0.000023156	-----	34
----	0.000010325	-----	35
----	0.0000044738	----	36
`----	0.0000018851	----	37

The `clean` function here trims off branches with weights less than the given threshold 10^{-6} to make it easier to view and interpret the Kind. (You can just view `E64` to compare.) We see that the largest weights are in the range 12 to 19, and

```
pgd> E(E64)
63/4
```

is our best prediction of the number of edges.

With some mathematical reasoning that we will derive later, we can go further and derive the *exact* Kind for `edge_count(random_graph(n, p))` for any n and p . Try `exact_edge_count(n,p)` to see this Kind, e.g., compare `exact_edge_count(64, '1/128')`. For very large n and very small p , we can also get an excellent approximation to `exact_edge_count(n,p)` that is quite fast.

Another approach we can use get results for large Kinds is *sampling*. We can compute transformed FRPs fairly easily, so by running demos of the transformed FRP, we get an empirical approximation of the Kind. As an example, compare the following demo with the exact Kind:

```
pgd> FRP.sample(1_000_000, is_connected(random_graph(6)))
```

```
Summary of Output Values
+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| 0      | 185191 | 18.52% |
| 1      | 814809 | 81.48% |
+-----+-----+-----+
```

```
pgd> kind(random_graph(6)) ~ is_connected
,---- 0.18506 ---- 0
<> -|
`---- 0.81494 ---- 1
```

Sampling gives us a numeric approximation of the Kind. The sample will always be somewhat inaccurate, but we can quantify that inaccuracy enough to make it useful.

We can answer the question of whether a graph is a tree even for very large n in the special case where $p = 1/2$. The function `exact_is_tree` computes that Kind for each n .

RANDOM IMAGES. Our next example considers FRPs that generate random binary images. Such an image is a rectangular grid of “pixels” whose value can be either 0 or 1. We will focus on 32×32 images, though the logic of the example works with any size. In the playground, load the tools needed for this example with

```
pgd> from frplib.examples.random_images import (
...>     random_image, as_image, add_images, show_image,
...>     image_distance, closest_image_to,
...>     erode, dilate, max_likelihood_image,
...>     ImageModels, pixel0, pixel1
...> )
```


or if you prefer to avoid typing, just do

```
pgd> from frplib.examples.random_images import *
```


As with the random graphs example, we will show the values of our FRPs as images, but in the playground you will see tuples, as described below. You can use the `show_image` function on a tuple or image FRP to pop up a view of the image in a browser window.

The function `random_image` is an FRP factory that returns an FRP representing a random binary image. (The default is a 32×32 image.)


```
pgd> random_image()
```

An FRP with value 


```
pgd> random_image(p='1/8')
```

An FRP with value 

```
pgd> random_image(p='1/6', base=ImageModels.image('P'))
```

An FRP with value 

```
pgd> random_image(p='1/8', base=ImageModels.image('minus'))
```

An FRP with value 

```
pgd> random_image(p='1/8', base=ImageModels.image('E'))
```

An FRP with value



```
pgd> random_image(p='1/8', base=ImageModels.image('blocks'))
```

An FRP with value



```
pgd> my_image = as_image(pixel0 * 416 + pixel1 * 64 + pixel0 * 64 +  
...>      pixel1 * 64 + pixel0 * 416)
```

```
pgd> random_image(p='1/4', base=my_image)
```

An FRP with value



```
pgd> ImageModels.register_image('my_image', my_image) # save image
```

These FRPs represent binary images with a random scattering of black pixels superimposed on a base image. Black pixels have a value 1 and white pixels a value 0. We can think of the images as pixelwise sum of a base image and random “noise”, where pixels add with an exclusive-or: $1 \oplus 0 = 1 = 0 \oplus 1$ and $0 \oplus 0 = 0 = 1 \oplus 1$. (Note the last equality.) The `base` argument to `random_image` sets the base image (by default all white). You can use pre-defined images in the `ImageModels` object, define your own with `as_image`, or add custom images to `ImageModels`. The `p` argument to `random_image` determines the intensity of the noise, with larger p making noisy pixels more prevalent.²⁹ The setting of `my_image` above specifies the pixel values left-to-right then top-down: 416 0’s followed by 64 1s then 64 0s then 64 1s and 416 0s.

²⁹Remember that you can use `help` in the playground to get full documentation on all these functions.

These FRP’s values are tuples of dimension $2+1024$, with the first two components the image width and height followed by 1024 0s and 1s arranged row-wise from the top left to the bottom right of the image. The Kinds of these FRPs have size 2^{1024} , so we will not be computing them directly. But we can use mathematical and computational reasoning to operate on the Kinds that we need. (As all the images here have Kinds of large size, I will elide the “It may be slow to evaluate its kind” warnings in playground output in this example.)

The example code defines a variety of statistics that operate on images.

```
pgd> black_pixels(random_image()) # Number of black pixels in image
```


An FRP with value <489>

```
pgd> clockwise(ImageModels.image('F'))
```



```
pgd> noisyF = random_image(p='2/15', base=ImageModels.image('F'))
```



An FRP with value

```
pgd> reflect_image_vertically(noisyF)
```



An FRP with value

```
pgd> largest_cluster_size(random_image(p='1/3'))
```

An FRP with value <98>

where `ImageModels.image('F')` returns a pre-defined image that happens to look like an F.

Our goal in the remainder of this example is to develop a way to reconstruct an underlying image from a noisy observed image. We will build an FRP that represents a random image with an *unknown* base image, but we *assume* that the base image belongs to a known collection of possible base images that we call a *model*. We will use statistics here as procedures for reconstructing the unknown base image. These statistics have type $2 + \text{width} * \text{height} \rightarrow 2 + \text{width} * \text{height}$ (e.g., $1026 \rightarrow 1026$) and map an observed, noisy image to an image that is an estimated reconstruction of the unknown base image. Our random image FRP represents the data, and the FRP transformed by these “reconstruction” statistics represent the inferred base image.

The `ImageModels` object has various methods for working with random image models. A model is specified by a set of *candidate* base images and a parameter $0 \leq p \leq 1$ that determines noise prevalence. When we observe an image generated under a particular model, one of the candidate base images is selected and noise with the specified prevalence is superimposed. The function `ImageModels.observe` returns an FRP representing a random image with a specified model. Several sets of candidate base images have been pre-defined with short identifiers. You can list the identifiers with `ImageModels.models()` and list a model’s base images with `ImageModels.model(id)`. `ImageModels.observe` accepts a model identifier or a list

of images in the first argument. It uses $p = 1/8$ by default, but you can override this with the named `p` argument. You can also register new sets of base images with `ImageModels.register_model`.

```
pgd> ImageModels.model('efh')
```



```
pgd> data, truth = ImageModels.observe('efh', p='1/4')
```

```
pgd> data
```

An FRP with value



We first see the three base images that might underlie our data, but we do not know which one it is. `ImageModels.observe` returns both an FRP (`data`) representing the observed data – a random image that superimposes noisy pixels on an unknown base image from the models – and the true base image (`truth`) on which the data is based. In practice, `truth` would be *unknown*, but for our purposes here, it is convenient to have access to it so that we can evaluate our procedure. Our goal is to devise a statistic that transforms the FRP `data` to an FRP whose value is an image as close as possible to the unknown `truth`.

From the image shown here, you can probably guess the unknown `truth`, but that's ok. The methods described here apply even when it hard to distinguish between the model images. Using an easier problem like this will clarify the ideas.

Puzzle 18. Create two or more images with `as_image` and use them to register a new model in `ImageModels`. Generate data from this model and look at the images.

Our first reconstruction statistic will be an algorithm to “de-noise” the image. It operates on the image by eliminating pixels that look like noise but retaining those that do not. The idea is that the de-noising statistic will preferentially remove the scattered blocks from noise while retaining significant structure from the base image. This procedure neither accounts for nor requires knowledge of the possible base images. To define our statistic, we need to define two operations on images: image *dilation* and image *erosion*. For both these functions, we think of an image's

pixels as lying on the grid of points in the plane with integer coordinates. If \mathcal{I} is a binary image, we can think of it as a set of integer coordinates at which there is a black/1 pixel. The dilation of an image \mathcal{I} with respect to a *dilation element* \mathcal{D} – a small set of integer coordinates – has black/1 pixels at coordinates³⁰

$$\text{dilate}_{\mathcal{D}}(\mathcal{I}) = \{ \langle x+i, y+j \rangle \mid \langle x, y \rangle \in \mathcal{I} \wedge \langle i, j \rangle \in \mathcal{D} \}. \quad (2.3)$$

Put another way, we overlay a copy of the dilation element at every black pixel in the image. For example, using the default dilation element $\mathcal{D}_0 = \{ \langle i, j \rangle \mid i, j \in \{-1, 0, 1\} \}$, a small square centered at 0, we have



where the dilated image is 34×34 . Dilation spreads out every black pixel in an image with a copy of the dilation element, so the dilated image is bigger by two pixels in each dimension.

Erosion is the dual operation to dilation. We specify an *erosion element* \mathcal{E} – again, a small set of integer coordinates – and the resulting image has black/1 pixels at coordinates

$$\text{erode}_{\mathcal{E}}(\mathcal{I}) = \{ \langle x, y \rangle \mid \langle x+i, y+j \rangle \in \mathcal{I} \text{ for all } \langle i, j \rangle \in \mathcal{E} \}. \quad (2.4)$$

We include all the pixels at which a copy of the erosion element is completely contained in the image. For example, using the default erosion element $\mathcal{E}_0 = \{ \langle i, j \rangle \mid i, j \in \{-1, 0, 1\} \}$, a small square centered at 0, we have



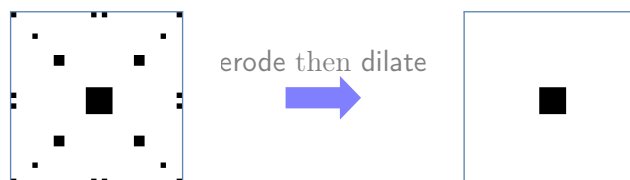
where the eroded image is 30×30 , two pixels smaller than the original image in each

³⁰In defining a set, the \mid is read “such that” or “given” and the \wedge is logical-and. The condition after the \mid constrains the elements of the set.

dimension. Try eroding and dilating a few of the pre-defined images to get a feel for what these do.

Our statistic for reconstructing the base image from the data is to first apply `erode` (with the default erosion element) and then to apply `dilate` (with the default dilation element). In the playground, `erode` and `dilate` are statistic factories that take the erosion and dilation elements³¹ and return the corresponding statistics. So, using the default elements, we write our “de-noising” statistic as `denoise = erode() ^ dilate()`. First we erode and then we dilate.


³¹Both accept instead an integer s , indicating a square element with each coordinate in $-s..s$.




Notice that this statistic keeps the resulting image the same size as the original. It eliminates the small blocks of black pixels but keeps the larger central block as is.

Let’s try it on a random image.

```
pgd> data
```


An FRP with value 

```
pgd> denoised = denoise(data)
```

An FRP with value 

```
pgd> reconstructed = closest_image_to(denoised.value, ImageModels.model('efh'))
```

```
pgd> reconstructed
```

An FRP with value 

```
pgd> image_distance(reconstructed, truth)
```


```
0
```

Here, `image_distance` counts the number of pixels in which `reconstructed` and `truth` differ. In `denoised`, we have successfully eliminated the noise pixels and only slightly degraded the base image. Our guess of the unknown base image is the image in our model that is closest to the value of `denoised`. This image, `reconstructed`,


turns out to be correct in this case, so we have successfully reconstructed the unknown image.

The `denoised` image in this case loses significant structure in the F, but keep in mind the image is 32×32 and the dilation and erosion elements are 3×3 with substantial noise. If we increase the resolution of the image and reduce the noise a bit, the denoising shows the structure of the F clearly.

```
pgd> hi_res, _ = ImageModels.observe([ImageModels.image('F#')])
```


An FRP with value 

```
pgd> denoised = denoise(hi_res)
```

An FRP with value 

```
pgd> denoise2 = erode() ^ dilate(2)
```

```
pgd> denoised2 = denoise2(hi_res)
```

An FRP with value 

In the last image, we dilate with a larger element to “fill in the holes.”

Puzzle 19. In `denoise`, we first erode then dilate. What happens if we first dilate then erode? Try it on a few noisy images.

We can encapsulate this procedure into a single statistic. Because the procedure depends on the models and the value of p , we write this as a statistic factory.

```
def reconstruct_image(model_id, denoiser=erode() ^ dilate()):
    """A statistics factory for reconstructing an unknown image from noisy data.

    Parameters
    -----
    model_id: int | str - an identifier for the model holding candidate base images
    denoiser: Statistic - a denoising statistic mapping image to image

    """
```

```

model_images = ImageModels.model(model_id)

@statistic
def reconstruct(observed_image):
    denoised_image = denoiser(observed_image)
    return closest_image_to(denoised_image, model_images)

return reconstruct

```

This might seem odd at first because we have a function that returns another function (a statistic), but all it does is let us set up the context in which the desired statistic can be defined based on the parameter (`model`) that we pass. (The statistic has access to all the local variables defined in the outer function.) Above, we used the equivalent of the statistic `reconstruct_image('efh')`.

To evaluate how well this procedure reconstructs the unknown image, we can repeatedly clone the data FRP, apply our statistic, and see how close our reconstruction is to the truth. We embody this with a single function:

```

def simulate_denoise(
    model_id, denoiser,
    p='1/8', observations=10_000
):
    """Evaluates denoising statistic on repeated observations from a model.

    Parameters:
        + model_id - identifier of pre-defined model in ImageModels
        + denoiser: Statistic - a denoising statistic mapping image to image
        + p: ScalarQ - noise prevalence (0 <= p <= 1), numeric or string
        + observations: int - number of observed images to generate

    Returns a pair of numbers giving (i) the proportion of incorrect
    reconstructions over all observations, and (ii) the average
    distance between truth and reconstruction over all observations.

    """

```

```

prop_wrong = 0
score = 0
for _ in range(observations):
    data, truth = ImageModels.observe(model_id, p=p)
    reconstructed = denoiser(data)
    distance = image_distance(reconstructed.value, truth)
    score += distance
    prop_wrong += (distance > 0)  # 0 if correct, 1 if not
return (prop_wrong / observations, score / observations)

```

Now, we can evaluate our procedure with different models and different noise specifications. We want a smaller number for both criteria. For instance:

```

pgd> reconstruct = reconstruct_image('efh')
pgd> simulate_denoise('efh', reconstruct)
(0.1831, 8.3244)
pgd> simulate_denoise('efh', reconstruct, p='1/32')
(0.008, 0.24)
pgd> simulate_denoise('efh', reconstruct, p='1/4')
(0.5667, 43.5033)

```

In the first case, reconstruction is correct about 82% of the time and even when they are not, it differs by only about $45 \approx 8.3244/0.1831$ pixels on average from the true image.³² As the noise level gets smaller or larger, the reconstruction performance gets better or worse, as we would expect. Try it yourself with several different possibilities.

Puzzle 20. Briefly explain what you take from these simulation results.

Our second statistic for reconstructing the unknown base image uses a different approach: over all candidate base images and all values of p , we choose that which *makes the image we observed as likely as possible* using the *Kind* of the data FRP. If we look at the difference between the observed image and a particular candidate base image, what should we see? If the candidate image is the true base image – the one used to generate the data – what is left should look like noise (produced with some value of p). But if the candidate image differs from the true base image, we will see noise pixels *plus* excess black/1 pixels from the difference between the candidate and

³²The distance is zero correct, so the average distance is $8.3244 = 0.1831d + 0.81790$, where d is the average distance for incorrect cases.

true base images. Combining two binary images with a pixelwise exclusive-or sets only pixels where the two images differ. If we combine a candidate base image with the observed image in this way, we will see a purely noise-like image when we have the right candidate and noise plus something more otherwise. So, we use the *Kind* of the noise FRP to assess how well each candidate base image “fits” the observed data, and pick the best fitting choice. Although these Kinds have large size, we can use mathematical reasoning to compute this efficiently.

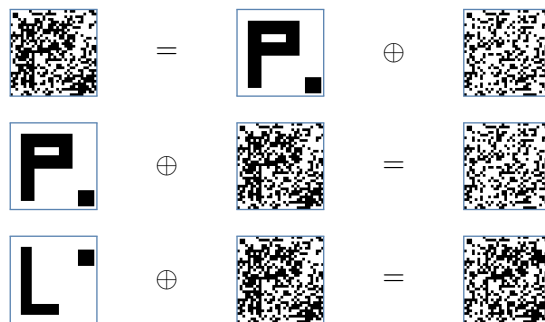


FIGURE 19. (Top) The observed image can be expressed as a combination (pixelwise exclusive-or) of the (unknown) base image and a noise image. (Middle) Combining the correct base image with the observed image gives us noise. (Bottom) But using the wrong base image gives us noise plus an excess of pixels where the candidate base image differs from the truth, which is visible in the rightmost image.

It will be helpful to formalize this one step further. For any base image \mathcal{J} , define the statistic $\psi_{\mathcal{J}}$ that maps an image \mathcal{Z} (of the same size as \mathcal{J}) to the image that has a 1 at any pixel where \mathcal{J} and \mathcal{Z} disagree and a 0 at any pixel where they agree. This is just the pixelwise exclusive-or of the two images, $\mathcal{J} \oplus \mathcal{Z}$; it shows us where \mathcal{Z} differs from \mathcal{J} .

Let Y be the data FRP that represents the observed image and suppose that it was generated with base image \mathcal{I} and noise parameter $0 \leq p \leq 1$. Then, the FRP $Z = \psi_{\mathcal{I}}(Y)$ represents an image of *purely noise*, as would be produced by `random_image(p)`. That is, if \mathcal{Y} is the value of Y (i.e., Y ’s output image) and \mathcal{Z} is the value of Z , then $\mathcal{Y} = \mathcal{I} \oplus \mathcal{Z}$. This relationship is illustrated in the top panel of Figure 19.

Because $\mathcal{I} \oplus \mathcal{I}$ is the empty image, it follows as well that $\mathcal{Z} = \psi_{\mathcal{I}}(\mathcal{Y})$. That is, if we take the difference of \mathcal{Y} and \mathcal{I} , we get $\mathcal{I} \oplus \mathcal{Y} = \mathcal{Z}$, the noise image. This is

illustrated in the middle panel of Figure 19.

However, if we use the *wrong* base image $\mathcal{J} \neq \mathcal{I}$ for the latter operation, we get noise plus parts of the base images. Specifically, the value of $\psi_{\mathcal{J}}(Y)$ is $(\mathcal{J} \oplus \mathcal{I}) \oplus \mathcal{Z}$. The $\mathcal{J} \oplus \mathcal{I}$ part is the difference between the base image we used and the true base image that generated the data. This will add “excess” pixels to the result. This is illustrated in the bottom panel of Figure 19.

We can use these relationships to choose a base image and a value of p . Assume that $p \leq 1/2$; this is not essential but it makes sense in practice. For candidate base image \mathcal{J} to be the true base image, then Z would need to have the same value as $\psi_{\mathcal{J}}(Y)$. *The weight of this value in the Kind of Z quantifies how likely we are to see that value.* The Kind of Z depends on p , so for each \mathcal{J} and each p , we get a score – the weight on the branch for value $\psi_{\mathcal{J}}(\mathcal{Y})$ in the Kind of Z . We choose the \mathcal{J} and p pair that maximizes this weight.

Formally, we define our reconstruction procedure as a statistic φ . Let $K_p(z)$ be the weight associated with value z in `kind(random_image(p))`, the Kind of noise generated with parameter $0 \leq p \leq 1/2$. The statistic φ takes an image \mathcal{Y} and chooses candidate base image \mathcal{J} and noise parameter p to *maximize* $K_p(\phi_{\mathcal{J}}(\mathcal{Y}))$, the weight associated with the “noise” that we see. This reconstructs the image by making the data we observed maximally likely.

In the playground module `frplib.examples.random_images`, the statistic φ is defined with `max_likelihood_image`. Like `reconstruct_image` earlier, this is a statistic factory that returns a statistic for the given model identifier (and an optional indication of whether to include the estimated p at the end of the returned tuple). Using `data` and `truth` generated earlier, we have

```
pgd> ml_image = max_likelihood_image('efh', return_p = True)
pgd> *max_liked, p_hat = ml_image(data).value
pgd> image_distance(max_liked, truth)
0
pgd> p_hat
0.131      # true value is 0.125
```

Remember that this method does not *know* the value of p ; it estimates it purely from the data. We can use the same type of simulations as earlier to evaluate this procedure.

```

pgd> ml_recon = max_likelihood_image('efh')
pgd> simulate_denoise('efh', ml_recon)
(0.0, 0.0)
pgd> simulate_denoise('efh', ml_recon, p='1/32')
(0.0, 0.0)
pgd> simulate_denoise(3, ml_recon, p='1/4')
(0.002, 0.06)

```

This performs very well, perfectly reconstructing in this simple model with low to moderate noise level p . This works because we have the excess pixels from the differences among the three candidate images are very unlikely to occur by chance. Try it yourself with several different possibilities.

2.5 frplib Statistics: Builtins, Factories, and Combinators

In this section, we examine some of the built-in tools for working with statistics in the playground. The playground defines a wide array of statistics and statistic factories to represent commonly-used operations and algorithms. It also offers tools, called statistic combinators, that combine statistics into new statistics in useful ways. Statistic expressions are a frequently used combinator that allows us to build custom statistics dynamically. We will also discuss how to define statistics with Python code, following up on the examples of this we have seen so far. Remember that you can type `help(s)` for any statistic `s` to learn about that statistic and call `info()` with argument `'statistics'`, `'statistics-builtin'`, `'statistic-factories'`, or `'statistic-combinators'` for documentation on various aspects of statistics in the playground. The `frplib` cheatsheet may also be helpful.

For these illustrations, we will define two FRPs: `D` represents the roll of five, balanced six-sided dice and `X` represents a random point in space as in Example 2.2. Do not worry about the definitions here; just enter them and follow along from there.

```

pgd> D = frp(uniform(1, 2, ..., 6)) ** 5
pgd> X = frp(uniform(-1,0,1)) ** 3
pgd> D
An FRP with value <5, 3, 1, 3, 5>. (It may be slow to evaluate its kind.)
pgd> X
An FRP with value <-1, 1, 0>

```

Look at your values of **D** and **X**, which will likely be different than those shown here, in preparation for looking at various transformed FRPs derived from them as you follow along. (The playground will warn you that **D**'s Kind may be slow to evaluate, though it's not bad at all; I will elide such warnings in the output below.)

The first group of built-in statistics are commonly used “summary statistics” that reduce a tuple to a scalar summary. Examples include **Sum**, **Product**, **Max**, **Min**, and **Mean** that compute, respectively, the sum, product, maximum, minimum, or arithmetic average of a tuple's components. Look at the sum of the five dice, the maximum of the five dice, and the product of the five dice:

```
pgd> Sum(D)
An FRP with value <17>.
pgd> Max(D)
An FRP with value <5>.
pdd> Product(D)
An FRP with value <225>.
```

The FRP representing the value of the first roll is a scalar

```
pgd> D1 = D[1] # Or equivalently: Proj[1](D), D ^ Proj[1]
pgd> D1
An FRP with value <5>
```

The playground also defines scalar statistics for common numerical functions. The names of these statistics are all capitalized, for instance: **Abs** for the absolute value, **Sqrt** for square root, **Exp** for the exponential, **Log**/**Log2**/**Log10** for natural/base 2/base 10 logarithm, **Sin**/**Cos**/**Tan** for trigonometric functions, and more.

```
pgd> Sqrt(D1)
An FRP with value <2.236067977499789696409173669>
pgd> Log2(D1)
An FRP with value <2.321928094887362347870319429>
pgd> Sqrt(Product(D)) # Or equivalently: D ^ Product ^ Sqrt
An FRP with value <15>
```

The last example shows that we can compose or chain transformations directly.

Two simple but useful statistics are the *identity function*, which returns its input as is, and the *constant function*, which returns the same output for all inputs.³³ The former is the statistic `Id`, and the latter is produced by the statistic factory `Constantly`, where `Constantly(v)` is the statistic that always returns value v .

³³See Table F.2.

```
pgd> Id(D)
An FRP with value <5, 3, 1, 3, 5>
pgd> D ^ Constantly(1, 2, 3)
An FRP with value <1, 2, 3>
```

We frequently want to use statistics that are built from simpler functions, and while we can define custom statistics as a Python function, it is often far easier to write a *statistic expression*. A key ingredient is the special statistic `__`. This acts as a “hole” in the expression to be filled by the argument to the statistic. Ordinary operations on statistics, including `__`, produce new statistics. Examples will clarify:

```
pgd> __
A Statistic '__' that represents the value given to the statistic.
It expects a tuple.
pgd> D1 ^ (6 * __)
An FRP with value <30>
pgd> D1 ^ Sin(FromDegrees(6 * __))
An FRP with value <0.5>
pgd> X ^ (2 * __ + 1)
An FRP with value <-1, 3, 1>
pgd> X ^ (2 * __ + (2, 0, 4))
An FRP with value <0, 2, 4>
pgd> X ^ (10 * __ + 2 ** __ + __ ** 2 + Abs(__))
An FRP with value <-7.5, 14, 2>
```

Here, `FromDegrees` is a statistic that converts degrees to radians, which is what the trigonometric statistics accept. Notice also in the several examples, the tuples add componentwise with the `+ 1` extending to a tuple of all 1’s. The last example shows that `__` can be used multiple times in one expression (here in a product, as an exponent, as a base, and as an argument to another statistic). We would typically write just `Abs` instead of `Abs(__)` in the last example, but both work. Both built-in

and custom statistics can be used in expressions. The use of the `^` operator in these examples is primarily for readability, but direct evaluation is also valid, e.g.,

```
pgd> (2 * __ + 1)(X)
An FRP with value <-1, 3, 1>
```

Expressions can use all the standard arithmetic and conditional operators. This does not include Python's Boolean operators `and`, `or`, and `not`, because these cannot be extended to handle custom objects. For these, we instead use the statistic combinators `And`, `Or`, and `Not`, as described below.

We have already seen the `Proj` factories for generating projection statistics. `Permute` is a similar factory that generates statistics that permute the tuples components. For instance, `Permute(2,1)` is the permutation that swaps the first two components of the tuple

```
pgd> Permute(2,1)(100,200)
<200, 100>
pgd> D ^ Permute(2,1)
An FRP with value <3, 5, 1, 3, 5>
```

The arguments to `Permute` use the cycle notation described in Section F.7 and in the `Permute` documentation. The related built-in statistics `Ascending` and `Descending` sort the tuple in increasing and decreasing order.

Three general statistic factories are used for converting a general function into a statistic: `statistic`, `scalar_statistic`, and `condition`.³⁴ The factories `statistic` and `scalar_statistic` wrap a Python function in a `Statistic` object, with the second distinguished only by configuring the statistic to have dimension 1.

³⁴All three of these can be used as *decorators* as well, as we will see.

```
pgd> range_of = scalar_statistic(lambda v: max(v) - min(v))
pgd> ascending = statistic(sorted)
pgd> range_of(7, 2, 0, 10)
<10>
pgd> ascending(7, 2, 0, 10)
<0, 2, 7, 10>
pgd> range_of(D)
An FRP with value <4>
```

```
pgd> ascending(D)
```

An FRP with value <1, 3, 3, 5, 5>

The `condition` factory converts a Python function or other statistic into a *condition* – a Boolean statistic; the return value of the function is treated as a Boolean and converted to 0 (false) or 1 (true). The predefined conditions `bottom` and `top` always return false and true, respectively.

```
pgd> isInteger = condition(lambda v: len(v) == 1 and isinstance(v[0], int))
```

```
pgd> isIntegerAlt = condition(lambda v: isinstance(v, int), codim=1)
```

```
pgd> isEven = condition(__ % 2 == 0)
```

```
pgd> isPositive = condition(Scalar > 0)
```

```
pgd> isEven(4)
```

```
<1>
```

```
pgd> isEven(0)
```

```
<0>
```

```
pgd> isPositive(-1)
```

```
<0>
```

```
pgd> isPositive(17.42)
```

```
<1>
```

```
pgd> isInteger(-4)
```

```
<1>
```

```
pgd> isInteger(4.2)
```

```
<0>
```

For `isInteger` and `isIntegerAlt`, we pass an ordinary (anonymous) Boolean function to `condition`. These are two alternative versions of the same condition. In the first case, the argument can be an arbitrary tuple, and we test its length and type; in the second case, we tell `condition` that the codimension must be 1, which ensures we get a scalar. Calling `isInteger` with a tuple of dimension > 1 will return false; calling `isIntegerAlt` with such a tuple will raise an error message. You can define the condition to get the behavior you want. The definitions of `isEven` and `isPositive` pass a Boolean *statistic* to `condition`. In the definition of `isPositive`, the use of `Scalar` in place of `__` makes the condition more robust by raising an error if a non-scalar is passed to the condition. (Otherwise, Python will gladly compare $(0,0,0) > 0$ and return true.)

This is not an issue for **Even** as the modulus operator `%` itself already requires a scalar. Often, conditions are combinations of Boolean statements with logical-and/or/not, and for this we use the **And**, **Or**, and **Not** combinators, all of which return conditions.

```
pgd> isDivisibleBy6 = And(__ % 2 == 0, __ % 3 == 0)
pgd> pos2_or_3 = And(Scalar > 0, Or(__ % 2 == 0, __ % 3 == 0))
pgd> isOdd = Not(isEven)
```

Try evaluating these statistics on various values to make sure you understand their meaning. We can also use the combinators **All** and **Any** that take a (scalar) condition and return a condition that is true if the given condition is true for every component or for some component of the value, respectively.

```
pgd> X ^ All(isPositive)
An FRP with value <0>
pgd> D ^ Any(Scalar >= 5)
An FRP with value <1>
```

Like **And**/**Or**/**Not**, **All** and **Any** return conditions, so they do not need to be wrapped in a call to **condition**.

The playground defines several other useful combinators, including the commonly used **ForEach**, **IfThenElse**, and **Fork**. **ForEach** takes a statistic and applies that statistic to each component of the given value, producing a new tuple. **IfThenElse** takes a condition and two statistics; if the condition is true for the given value it applies the first statistic else the second. (If a value v is given instead of a statistic in the second or third argument, it is equivalent to passing **Constantly**(v).) **Fork** takes one or more statistics and applies them all to the given value, concatenating their results into a tuple.³⁵ For example:

```
pgd> X ^ ForEach(5 * __ + 5)
An FRP with value <0, 10, 5>
pgd> X ^ IfThenElse(Sum >= 2, Max, Min)
An FRP with value <-1>
pgd> D ^ ForEach(IfThenElse(Scalar <= 3, 0, 2 * __ - 6))
An FRP with value <4, 0, 0, 0, 4>
pgd> D ^ IfThenElse(Proj[2] < 4, Proj[5], Proj[2])
```

³⁵See the fork operator \curlyvee in Section F.7.

An FRP with value <5>

```
pgd> D ~ Fork(Min, Mean, Max)
```

An FRP with value <1, 3.4, 5>

In the first case, the statistic transforms each component x to $5x + 5$, taking -1, 0, 1 to 0, 5, 10, respectively. In the second case, we take the maximum for points whose component sum is ≥ 2 else the minimum. In the third case, any components (die rolls) ≤ 3 are replaced by 0 and others x map to $2x - 6$, taking 4, 5, and 6 to 2, 4, and 6. In the fourth case, we use the fifth roll if the second is smaller than 4 and otherwise the second roll. And in the last case, we collect the minimum, mean, and maximum of the five dice rolls. Notice how we use statistics (like `Sum` or `Proj[5]` or `--` above) as part of the expressions to express complicated logic.

Keep in mind that the statistics created with these tools can be used in all the ways shown earlier. In particular, we can give them names, evaluate them as functions on values, and use them to transform Kinds. For example:

```
pgd> psi = IfThenElse(Proj[2] < 4, Proj[5], Proj[2])
```

```
pgd> psi(1,6,3,2,1)
```

<6>

```
pgd> kind(psi(D))
```

```
,---- 1/12 ---- 1
|---- 1/12 ---- 2
|---- 1/12 ---- 3
<> -|
|---- 3/12 ---- 4
|---- 3/12 ---- 5
`---- 3/12 ---- 6
```

These dynamic statistics are useful and convenient, but if the algorithm for computing a statistic is complicated, it can be easier to write custom, named statistics. To do this, we define ordinary Python functions and precede them with one of the *decorators* `@statistic()`, `@scalar_statistic()`, or `@condition`. If the Python function has multiple arguments without default values, that will specify the codimension of the statistic; if it has a single argument, it will accept an arbitrary tuple. The `codim` and `dim` arguments to the decorators can be used to specify the statistic's type as

well. For numeric FRPs, as we usually use, the functions should return a number (for dimension 1) or tuples of numbers. Regular Python tuples that are returned will be converted to the playground's `VecTuples` which offer some extra utility. Numbers returned can include $\pm\infty$, written as `infinity` and `-infinity`.

For example, consider two questions about D , which represents five dice rolls: How many rolls does it take until we see the first 6? How many rolls have the most common value seen among the five? We can define statistics to answer both questions. For the first question:

```
pgd> @scalar_statistic(description='counts rolls until a 6, or infinity if none')
...> def when_first_6(rolls):
...>     try:      # This will fail unless there is a roll of 6
...>         return 1 + rolls.index(6) # .index() of first 6 from 0
...>     except:   # There is no roll with value 6, do this
...>         return infinity
pgd> when_first_6
A Statistic 'when_first_6' that counts rolls until a 6, or infinity if none.
It expects a tuple and returns a scalar.
pgd> when_first_6(D)
An FRP with value infinity
pgd> when_first_6(clone(D))
An FRP with value <3>
```

For the second question:

```
pgd> @statistic
...> def most_common_count(rolls):
...>     "returns number of rolls with most common value"
...>     counts = [0] * len(rolls)
...>     for roll in rolls:
...>         counts[roll] += 1
...>     return max(counts)
pgd> most_common_roll
A Statistic 'most_common_count' that returns number of rolls with most common value.
It expects a tuple.
```

```
pgd> most_common_roll(D)
An FRP with value 2.
```

The description is optional and is taken from the function's docstring if provided.

As another example, X represents a point inside a cube, and we might ask whether the points is a corner, edge, face, or center. (See Example [2.2](#).)

```
pgd> @statistic
...> def classify_point(x):
...>     "determines type of cube point (0=corner,1=edge,2=face,3=center)"
...>     distance_from_origin, = Norm(x)
...>     if distance_from_origin >= numeric_sqrt(3):
...>         return 0 # corner
...>     if distance_from_origin >= numeric_sqrt(2):
...>         return 1 # edge
...>     if distance_from_origin >= 1:
...>         return 2 # face
...>     return 0 # center
pgd> classify_point(X)
An FRP with value <1>
```

Here we use the built-in statistic `Norm` to compute the distance of the point from the origin and deconstruct the tuple to extract the first value.

As an example showing how to use arguments in custom statistics, consider

```
pgd> @statistic(dim=2, arg_convert=numeric.as_real)
pgd> def quadratic_roots(a, b, c):
...>     "returns roots of quadratic  $a x^2 + b x + c$ "
...>     if is_zero(a):
...>         root = -b/c
...>         return (root, root)
...>     center = -b / (2 * a)
...>     disc = numeric_sqrt(center * center - c / a)
...>     return (center - disc, center + disc)
pgd> quadratic_roots(1, 0, -1)
<-1, 1>
```

```
pgd> quadratic_roots(1, 2, 1)
<1, 1>
```

Here, the `arg_convert` argument ensures that all the arguments are converted to high-precision real quantities, and the function `is_zero` checks if a quantity is zero within numerical precision.

Puzzle 21. Create a statistic like `most_common_count` that produces the *value* of the most common roll *and* the count of how many times it appeared.

Puzzle 22. Create a statistic that answers the question of whether either pattern 1, 2, 3 or pattern 4, 5, 6 occurs in three successive rolls.

In addition to the playground’s help/info system and documentation, the “Playground Overview” on page 106 summarizes the most commonly used built-in statistics, factories, and combinators. A Playground Cheatsheet is also available.

Checkpoints

After reading this section you should be able to:

- Define a statistic and give several examples of useful statistics.
- Explain how to transform an FRP or Kind using a statistic.
- Explain what it means for a statistic and FRP/Kind to be compatible.
- Use the playground to construct statistics.
- Use the playground to transform an FRP or Kind using a statistic.
- Define the components of an FRP.
- Use projection statistics (via `Proj`) to find the Kind of an FRP component and to construct the FRP for a component.
- Find help and documentation on built-in statistics, factories, and combinators in the playground.
- Use built-in statistics, factories, and combinators in the playground.

Playground Overview

Most operations in the playground can be categorized as either [factories](#), [combinators](#), or [actions](#). Factories create things, combinators combine existing things into a new thing, and actions use things to produce an effect. Here we list some of the most commonly used of these; see the `frplib` help and cheatsheet for more.

Kind Factories

`kind` – constructs a Kind from a string, an FRP, or another Kind.
`conditional_kind` – constructs a conditional Kind from a dict or function
`fast_mixture_pow` – computes `mstat(k ** n)` efficiently
`constant` – the Kind of a constant FRP with specified value
`uniform` – the Kind with specified values and equal weights
`either` – `either(a,b,w=1)` has values `a` and `b` with weights `w` and `1`
`weighted_as` – specified weights on arbitrary values
`weighted_by` – weights on values determined by a general function
`evenly_spaced`, `integers`, `symmetric`, `linear`, `geometric` – kinds on specified values with patterns of weights
`subsets`, `without_replacement`, `permutations_of`, `ordered_samples`
`arbitrary` – the Kind with specified values and symbolic (unspecified) weights

FRP Factories

`frp` – constructs an FRP from a Kind or clones another FRP.
`conditional_frp` – constructs a conditional FRP from a dict or function
`shuffle` – an FRP that shuffles a given sequence

Kind and FRP Combinators

`^` operator – `a ^ stat` and `stat(a)`, transform `a` with statistic
`*` operator – `a * b` is the independent mixture of `a` and `b`
`**` operator – `a ** n` is the independent mixture of `a` with itself `n` times
`>>` operator – `a >> b` is the mixture with mixer `a` and target `b`
`|` operator – `a | c` is the conditional of `a` given the condition `c`
`//` operator – `b // a` (read “`b` conditioning on `a`”) is equivalent to
`a >> b ^ Proj[-b.dim, -b.dim+1, ..., -1]`

Playground Overview (cont'd)

Statistic Factories

`statistic`, `condition`, `scalar_statistic` – convert a function into a statistic
`Constantly` – a statistic that always returns the same value
`Proj` – produces a projection statistic on the given indices
`Permute` – produces a permutation statistic with the given permutation

Statistics

`--`, `Scalar` – stands for the value passed in, the latter forces a scalar
`Sum`, `Product`, `Min`, `Max`, `Count` – arithmetic operations on value's components
`Mean`, `StandardDeviation` – statistical summaries of a value's components
`Abs`, `Dot` – absolute value/norm and dot product with a specified vector
`Diff` and `DiffS` – successive differences of the values components
`Exp`, `Log`, `Log2`, `Log10`, `Sin`, `Cos`, `Tan`, `Sqrt`, `Floor`, `Ceil`, `NormalCDF` – scalar mathematical functions
`top`, `bottom` – statistics that always return true and false

Statistic Combinators

`^` – `s1 ^ s2` (“s1 then s2”), equivalent to `s2(s1)`
`@` – `stat @ X` is like `stat(X)` but passes `X` to a following conditional
`And`, `Or`, `Xor`, `Not` – logic operators
`Fork` - `Fork(f1,f2,...,fn)` applies each `fi` to its input
`ForEach` – apply a statistic to each component of a value
`IfThenElse` – if a condition is true, apply one statistic else another.

Actions

`symbol`, `symbols` – create symbolic quantities with given names
`clone` – create copy of an FRP or conditional FRP with its own value
`unfold` – unfold a canonical Kind tree
`clean` – remove branches with numerically negligible weights
`FRP.sample` – activate clones of a given FRP

Utilities

`dim`, `codim`, `size`, `values` – get properties
`irange`, `index_of` – inclusive integer ranges and index finding
`identity`, `const`, `compose` – useful functions
`frequencies` – compute frequencies of values in a sequence
`as_quantity`, `qvec` – convert to quantity (numeric/symbolic) or quantity vector
`numeric_abs`, `numeric_log`, `numeric_exp`, `numeric_sqrt` – scalar ops

107

Help

`info` – frplib specific help
`help` – built-in python help

3 Equivalent Kinds and Canonical Forms

Key Take Aways

Kinds are described by complete trees, with values at the nodes and weights on the edges. In our empirical study of FRPs and Kinds, we saw that structurally different trees can produce the *same predictions*. Here, we explore conditions in which two distinct Kinds give equivalent descriptions of an FRP.

Two Kinds k and k' are **equivalent** when (i) they have the *same sets of values* on their leaves, and (ii) FRPs with Kind $\psi(k)$ and $\psi(k')$ have the *same risk-neutral price* for any compatible, scalar statistic ψ . Given FRPs with equivalent Kinds k and k' , one cannot distinguish whether the FRP has Kind k or k' just by observing the FRPs values, even in the aggregate. The bottom line is that FRPs with equivalent Kinds are completely interchangeable.

Kinds that differ only in the order of branches at a node are equivalent. Kinds that differ only in a constant scaling of the weights branching from any node are equivalent. And any Kind can be reduced to an equivalent Kind in *compact* form (i.e., width 1) using Algorithm COMPACT.

Every Kind tree has a **canonical form**, which can be obtained (Algorithm CANONICAL) by

1. Ordering the leaves from top to bottom in increasing lexicographic order.
2. At each non-leaf node of the tree, normalizing the weights on the edges branching from that node so that they sum to 1.
3. Reducing the tree to compact form using Algorithm COMPACT.

Every Kind is **equivalent to its canonical form. Two Kinds are equivalent if they have the same canonical form.**

Algorithms COMPACT and UNFOLD can be used to convert between a single-level compact form and a multi-level (in general) *unfolded* form.

An FRP produces a single value, fixed for all time once the button is first pushed. So how can we predict anything about its value? Fortunately, we have seen in our empirical investigations that we can make predictions *in the aggregate* by demoing many FRPs of the same *Kind*. The Kind represents an “ideal” version of the demo where we include *all* FRPs of the that Kind. What we see in a demo with a *finite*

number of FRPs will vary somewhat, but as we demo more and more FRPs of that Kind, the relative frequencies of the values we see in the demo will more closely match the weights in the Kind.

Kinds are described by weighted, complete trees, with values of the same dimension on the leaves and positive numbers for the weights. As we intuited in our explorations earlier, it is possible to have different trees that are *equivalent* in terms of the predictions they make about an FRP of that Kind. In this section, we take a closer look at when two Kind trees are equivalent and see how to convert among different representations of equivalent Kinds.

To illustrate equivalence, let us return to the market and use the `compare` task to run demos for two different Kinds in parallel.

```
mkt> compare 1_000_000 with kinds (<> 1 <0> 1 <1>) (<> 0.5 <0> 0.5 <1>).
```

Kind A

```
,----- 1 ----- <0>
<> -|
     `----- 1 ----- <1>
```

Kind B

```
,----- 1/2 ----- <0>
<> -|
     `----- 1/2 ----- <1>
```

Summary of Demo for Kind A

Values	Count	Proportion
0	499687	49.97%
1	500313	50.03%

Summary of Demo for Kind B

Values	Count	Proportion
--------	-------	------------

0	499629	49.96%
1	500371	50.04%

+-----+-----+-----+

The proportions here are not exactly the same, but the small differences are variations between the finite samples of FRPs in the two demos. The more demos we run, the closer the results for the two Kinds will become. These two Kinds are equivalent.

Consider also:

```
mkt> compare 1_000_000 with kinds
...> (<> 1 (<-1> 0.4 <-1, -15> 0.6 <-1, -5>)
...>      3 (<0> 1 <0, 10>)
...>      2 (<9> 1 <9, 12> 4 <9, 20> 5 <9, 32>))
...> (<> 1/15 <-1, -15> 1/10 <-1,-5> 1/2 <0,10>
...>      1/30 <9,12> 2/15 <9,20> 1/6 <9,32>).
```

```

Kind A
      ,----- 2/5 ---- <-1, -15>
,----- 1 ---- <-1> -|
|                   `----- 3/5 ---- <-1, -5>
|
|
|
<> -+----- 3 ---- <0>  -+----- 1 ----- <0, 10>
|
|
|
      ,----- 1 ----- <9, 12>
`----- 2 ---- <9>  -+----- 4 ----- <9, 20>
                   `----- 5 ----- <9, 32>

```

```

Kind B
,----- 2/30 ----- <-1, -15>
|
|----- 3/30 ----- <-1, -5>
|
|----- 15/30 ----- <0, 10>

```



```

<> -|
    |----- 1/30 ----- <9, 12>
    |
    |----- 4/30 ----- <9, 20>
    |
    |----- 5/30 ----- <9, 32>

```

Summary of Demo for Kind A

Values	Count	Proportion
<-1, -15>	66311	6.631%
<-1, -5>	100016	10%
<0, 10>	500824	50.08%
<9, 12>	33216	3.322%
<9, 20>	133040	13.3%
<9, 32>	166593	16.66%

Summary of Demo for Kind B

Values	Count	Proportion
<-1, -15>	66741	6.674%
<-1, -5>	99935	9.993%
<0, 10>	499578	49.96%
<9, 12>	33170	3.317%
<9, 20>	133211	13.32%
<9, 32>	167365	16.74%

Again, there are small variations between finite samples of FRPs, but these two Kinds are also equivalent.

We define the equivalence of two Kinds in terms of the risk-neutral prices of transformed FRPs with those Kinds. As discussed earlier, the risk-neutral price is

how much we would pay – ignoring our personal aversion or attraction to risk – for the value produced by an FRP and as such represents our best prediction of that FRP’s value.³⁶

³⁶We will see how to *find* these prices in Section 7; for now, we only need the idea.

Two Kinds K and K' are equivalent if two conditions hold:

1. every scalar statistic compatible with K is compatible with K' , and vice versa;
2. for every compatible scalar statistic ψ , we are indifferent to exchanging a fresh FRP with Kind $\psi(K)$ for a fresh FRP with Kind $\psi(K')$ with no money changing hands.

The first condition holds if and only if the two Kinds have the *same set of values on their leaves*. The second condition tells us that FRPs with Kind $\psi(K)$ and Kind $\psi(K')$ *have the same risk-neutral price*. That is, our predictions of these FRPs’ values are the same.

To understand the role of the statistics ψ here, think of the perspective from the last section: where each statistic corresponds to a question we might ask about the unknown value of the FRP with the transformed value answering that question. A scalar statistic corresponds to a question with a numerical answer. If we have FRPs X and X' with respective Kinds K and K' , then for any scalar statistic ψ , the values of $\psi(X)$ and $\psi(X')$ are the answers to the question for the values of X and X' . When K and K' are equivalent, it means that our *predicted answers* to the question are the same for *any meaningful question* we might ask.

Loosely speaking, two Kinds k and k' are equivalent when, before observing any FRPs’ values, we are indifferent to replacing any FRP of Kind k with an FRP of Kind k' and vice versa *no matter what transformation rule* our adversary chooses. If one Kind were easier to predict or tended to produce bigger payoffs or if we could *distinguish* the Kinds based on the values we see, then we would not be indifferent between them. Equivalent Kinds lead to FRPs that are *interchangeable*.

Definition 8. Two Kinds K and K' are **equivalent**, which we denote by $K \cong K'$, if both the following conditions hold:

1. K and K' have the same sets of values on their leaves.
2. For any compatible, scalar statistic ψ , FRPs with Kinds $\psi(K)$ and $\psi(K')$ have the *same risk-neutral price*.

Another way to think about this is that if you have a large demo of FRPs that all have either Kind K or K' , then you cannot distinguish between equivalent Kinds by looking at the values of these FRPs, even in the aggregate. Fresh FRPs with equivalent Kinds are thus interchangeable; we are indifferent to which we have as all our predictions about their values are the same.

Puzzle 23. A Kind k of the form

$$\langle \rangle \begin{cases} \text{---} a \text{---} \langle u \rangle \\ \text{---} b \text{---} \langle v \rangle \end{cases}$$

with $a, b > 0$ and $u \neq v$ has risk-neutral price $\frac{au+bv}{a+b}$, as we will see in Section 7.

Use this and the definition of equivalence to show that the following two Kinds are equivalent:

$$\langle \rangle \begin{cases} \text{---} 6 \text{---} \langle 0 \rangle \\ \text{---} 18 \text{---} \langle 12 \rangle \end{cases} \quad \langle \rangle \begin{cases} \text{---} \frac{1}{4} \text{---} \langle 0 \rangle \\ \text{---} \frac{3}{4} \text{---} \langle 12 \rangle \end{cases}$$

As a first step, write the tree for the transformed Kind $\psi(k)$ for an arbitrary, compatible scalar statistic ψ . You can do this without having a concrete expression for ψ ; the answer depends only on $\psi(u)$, $\psi(v)$, a , and b .

Equivalence (\cong) gives a binary relation on the set of Kinds, and in fact it is what we call an *equivalence relation*.³⁷ any Kind is equivalent to itself ($K \cong K$), the relation is symmetric ($K \cong K'$ if and only if $K' \cong K$), and the relation is transitive ($K \cong K'$ and $K' \cong K''$ implies $K \cong K''$). The relation \cong partitions the set of all Kinds into disjoint subsets of equivalent Kinds called *equivalence classes*. All Kinds in an equivalence class are equivalent to each other and are not equivalent to any Kind outside the equivalence class.

Condition 1 in the definition of equivalence is easy to check just by inspecting the trees for two Kinds. We look at the leaves and see if the sets of possible values are the same. Condition 2, however, seems harder to check, even with the formula for the risk-neutral price that we will derive later. How do we check it for all compatible, scalar statistics? Fortunately, because we have an equivalence relation, there is an easier way. We systematically choose one representative Kind for each equivalence

³⁷The notions of relation and equivalence relation are defined formally and discussed in Section F.8.

class, called the **canonical form** for Kinds in the equivalence class, and convert any Kind to its canonical form. Every Kind is equivalent to its canonical form, and thus by transitivity, any two Kinds with the same canonical form are equivalent.

Given a Kind tree, there is a simple algorithm to produce its canonical form. To motivate this, we consider three arbitrary choices we make in specifying a Kind: the order of branches at each node, the scaling of the weighs, and the width of the tree. Different choices in these dimensions lead to equivalent Kinds, so if we make a canonical choice in each, we get the canonical form.

First, when we display a Kind, we must specify the order of branches at each node, but this choice is arbitrary. For example, in Figure 20, the two Kinds differ only in branch order. They have the same values and for each value, the same weights along the path from root to leaf. If we run demos of both Kinds and tabulate the values in the market, the results do not depend on the branch order.³⁸ FRPs with these Kinds are indistinguishable in their behavior. So: *Kinds that differ only in the order of branches at a node are equivalent.*

³⁸Try this yourself, using compare in the market.

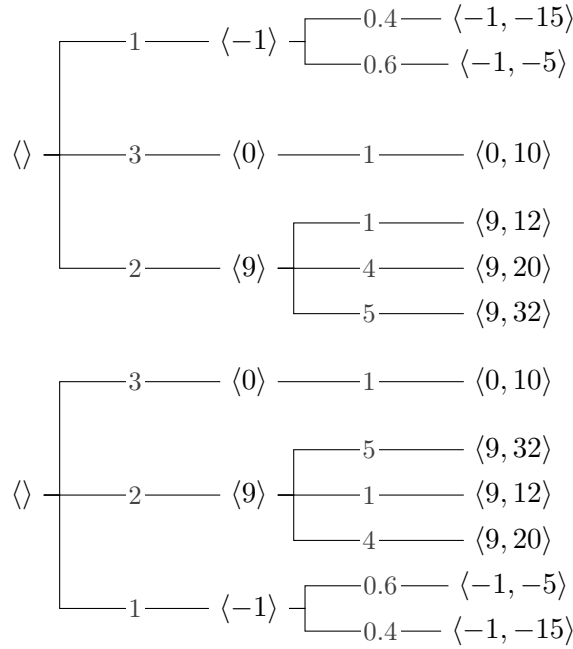


FIGURE 20. Two Kinds that differ only in the order of branches at some nodes.

We can choose a *canonical* branch order. Any choice will do, but it helps to be systematic. At each branching, order the nodes in increasing order of the last component in the value. Equivalently, we order the branches so that the leaf tuples are sorted from bottom to top in increasing lexicographic order (sort first by the first component, then by the second, and so forth). We are not required to use this order, but it provides a standard for comparison, as we will see below. For example, the top Kind in Figure 20 is in canonical branch order.

Another choice we have to make in specifying a Kind is the scaling of the weights. That is, we can multiply the weights at any branching by the same constant. Consider the Kinds in Figure 21. Are these distinguishable? Try this in the market using the `compare` task, as shown earlier. Can you tell these apart?

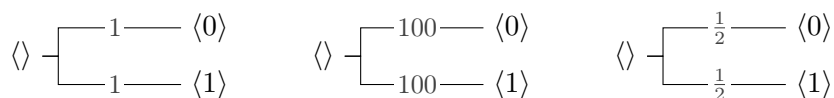


FIGURE 21. Three Kinds whose weights differ only by a constant multiplicative factor.

Short answer: no. If we scale all the weights branching from any node in a Kind tree by the same multiplicative factor, we get a new Kind whose demos will be indistinguishable from the original. Two Kinds are related in this way if the weights on the edges branching from some node w_1, \dots, w_m and w'_1, \dots, w'_m , satisfy: there is a $c > 0$ where $w_i/w'_i = c$ for every i . The constant can be different at each node, but all the branches emerging from a node must be scaled alike. So: *Kinds that differ only in a constant scaling of the weights weights branching from any node are equivalent.*

We make a canonical choice of scalings: *the sum of the weights emanating from any node should equal 1*. Again, we are not required to use this scaling, and it is sometimes convenient not to, but it serves as a standard to make comparison – and some other calculations – easier.

The last choice we make in presenting Kinds is the width of the tree (i.e., the number of levels) for Kinds of dimension > 1 . Having multiple levels in the tree emphasizes the sequential nature of the values generated and highlights the contingent choices made for each component of the generated value. This is often conceptually useful when building a model of a random process. At the same time, however, there is redundant information in the multi-level presentation. Figure 22 shows two

equivalent Kinds with different widths. The Kind shown on the left has the same width as its dimension; we call this the *unfolded* form. The Kind shown on the right has the same size, dimension, and values but has width 1; we call this the *compact* form. It might not seem obvious at first but the two are equivalent, and we can easily convert between them using Algorithms UNFOLDED and COMPACT below. So: *A Kind in unfolded form is equivalent to its compact form, and vice versa.*

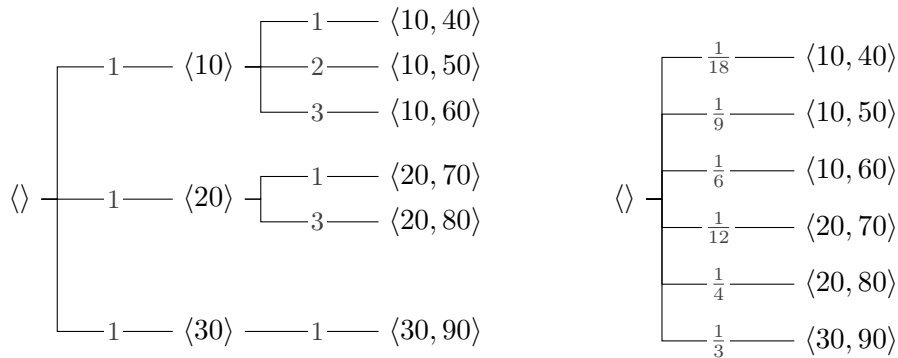


FIGURE 22. A Kind in two forms: unfolded and compact. Can you go from one to the other?

Our canonical choice is to show Kinds in compact form. Again, we are not required to use this choice – unfolded form can be illuminating – but it will be our default.

Combining these three transformations gives us a simple algorithm for converting any Kind to its **canonical form**. This is described by the following algorithm.

Algorithm CANONICAL

Given as input a Kind K , returns the canonical form of that Kind in three steps:

1. Order the leaves from top to bottom in increasing lexicographic order.
2. At each non-leaf node of the tree, normalize the weights on the edges branching from that node so that they sum to 1.
3. Reduce the tree to compact form using Algorithm COMPACT.

The result is the **canonical form** of Kind K .

The canonical form is a simple standard that makes it easy to compare and

manipulate Kinds and to identify equivalence. Other equivalent forms can also be useful, giving us insights about the random process or helping with calculations.

Every Kind is **equivalent** to its canonical form. Two Kinds are equivalent if they have the same canonical form.

So from here on, when we consider a Kind, we will effectively identify it with the class of Kinds that are equivalent to that tree. We treat the canonical form as a representative of this class and freely translate to other forms as needed.

Puzzle 24. Apply Algorithm CANONICAL to the left Kind in Figure 22. You should get the right Kind in that Figure.

Naming Convention.

This is a good point to establish a naming convention for FRPs and Kinds, to make it easier to reference them.

For FRPs, we will name them with *capital Roman letters* (like X, Y , and Z), sometimes with subscripts to indicate FRPs that are related in some way. Thus, we can name FRPs X, Y_1, Y_2, R, M, D_T and so forth. We often use integer subscripts to identify component FRPs of a multi-dimensional FRP. If we want to emphasize that a group of FRPs represent distinct FRPs with the same Kind, we will use the same base letter and wrap the subscripts with the index in brackets. For instance, a collection of four like-kinded FRPs $X_{[1]}, X_{[2]}, X_{[3]}, X_{[4]}$.

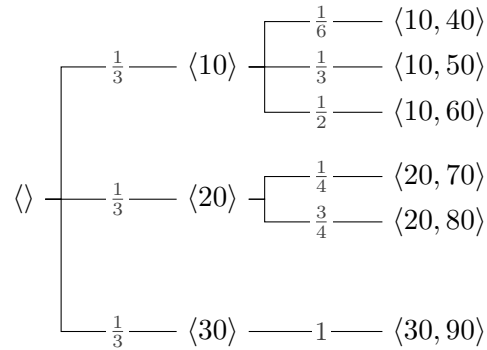
If X is an FRP, $\text{kind}(X)$ is its Kind, $\text{dim}(X)$ is its dimension, $\text{size}(X)$ is its size, and its set of values is $\text{values}(X)$.

For Kinds, we will name them with adorned letters like K, k, k', k_1, \dots , preferring to use the base letter k/K whenever possible.

Remember that, formally, $\text{kind}(X)$ refers to an equivalence class of trees, and we can display it with any of the equivalent trees in the class, with the canonical form by default.

Finally, we turn to the algorithms for compactifying and unfolding a Kind tree. Consider first the Kind at the left of Figure 22. We will carry out Algorithm COMPACT

in three steps. First, we normalize the weights so that for each non-leaf node, the branches coming from that node have weights that sum to 1. The tree becomes



We work one non-leaf node at a time, including the root. Typically, we would reduce fractions to lowest terms, but that is not always necessary or clarifying. Then, for each leaf node, we multiply the normalized weights along the path from root to leaf, recording the result. For instance, for the $\langle 10, 40 \rangle$ leaf node, we get $\frac{1}{3} \cdot \frac{1}{6} = \frac{1}{18}$; for the $\langle 20, 80 \rangle$ leaf node, we get $\frac{1}{3} \cdot \frac{3}{4} = \frac{1}{4}$; and so on. Creating a one level tree with the same leaf nodes and the weights corresponding to these products yields the compact form at the bottom of the Figure.

Algorithm COMPACT

Input: a Kind as an unfolded tree

Returns: the Kind in equivalent compact form.

Step 1. At each non-leaf node of the tree, normalize the weights on the edges branching from that node so that they sum to 1.

Step 2. For each leaf node of the tree, multiply together the weights along the path from the root to that leaf. Record the resulting product for that leaf.

Step 3. Create a single level tree with the same leaf nodes and set the weight for each leaf node to be the product you computed for that node in Step 2.

The resulting Kind tree is the compact form.

In the playground,³⁹ we can view and manipulate Kinds to understand this transformation. FRPs and Kinds can be assigned to variables for easy reference. Pick

³⁹When showing playground input and output, text from # to the end of a line is a comment for your benefit. You should not type or enter that.

a Kind, apply the algorithms, and then use commands like the following to see both forms.

```
pgd> practice_1 = '(<> 10 (<3> 1 <3, 2> 7 <3,3>)
...>                  11 (<30> 4 <30,0> 8 <30,2>))'
pgd> k1 = kind(practice_1)    # The kind specified by practice_1
pgd> unfold(k1)               # shows the unfolded form
pgd> k1                       # shows the k1's canonical form
```

The playground can do much more, as we will see in the next section.

Now let's go the other way, starting from the right tree Figure 22 and producing the left. The key to making this work is that each value generated at each level must be distinct. First, we normalize the weights as in the previous two algorithms, then we build the unfolded form *from the leaves up*.

The leaf nodes in the unfolded form will be the same as in the compact form. To get the nodes at the next higher level, we remove the *last* value in the list. When we do this, we get three $\langle 10 \rangle$'s and two $\langle 20 \rangle$'s, and because values must be distinct, we need to combine each of these sets into a subtree. Let's focus on the $\langle 10 \rangle$'s and the three nodes from which they come. These nodes will be grouped in a subtree with $\langle 10 \rangle$ at the branch. Add together the weights for these three nodes, yielding $\frac{1}{18} + \frac{1}{9} + \frac{1}{6} = \frac{1}{3}$. We carry the $\frac{1}{3}$ forward and divide each of the nodes' weights by $\frac{1}{3}$ to renormalize the sum to 1. Our subtree weights then become $\frac{1}{6}$, $\frac{1}{3}$, and $\frac{1}{2}$ respectively. Now we repeat the process with the node $\langle 10 \rangle$. We remove the last item from the list which gives the empty node; there are no repeats here. The $\frac{1}{3}$ that we carried over becomes the weight for that edge.

For completeness, let's do the other two cases. The leaf nodes that start with 20 have weights $\frac{1}{12}$ and $\frac{1}{4}$; adding these gives $\frac{1}{3}$. Renormalizing gives weights $\frac{1}{4}$ and $\frac{3}{4}$ for the subtree with nodes $\langle 20, 70 \rangle$ and $\langle 20, 80 \rangle$, carrying $\frac{1}{3}$ forward. Repeating for $\langle 20 \rangle$ brings us to the root, so that edge has weight $\frac{1}{3}$.

The node is $\langle 30, 90 \rangle$. We remove the 90, but there no duplicates and the weight is $\frac{1}{3}$, which normalizes to 1, carrying $\frac{1}{3}$ forward. Removing 30 brings us to the root with a weight of $\frac{1}{3}$. The result is as in the earlier Figure.

Algorithm UNFOLDED carries out these same operations for arbitrary Kinds.

Algorithm UNFOLDED

Input: a Kind as a compact (single level) tree

Returns: the Kind in equivalent unfolded form.

Step 1. Convert the compact tree to canonical form; in particular, normalize the weights in the input Kind so that they sum to 1. Call this T_0 .

Step 2. Define two kind-valued variables S and T . Initialize both to T_0

Step 3. While Kind S has dimension > 1 , do the following:

- i. Partition the leaf nodes S into disjoint sets $\mathcal{L}_1, \dots, \mathcal{L}_m$ (for some $m \geq 1$) of leaf nodes whose values are equal excluding the last element.
- ii. For \mathcal{L}_j with $j \in [1..m]$, do the following:
 - a. Let n_1, \dots, n_k be the leaf nodes in \mathcal{L}_j , with values of the form $\langle v_1, \dots, v_{d-1}, x_i \rangle$ $d = \dim(S)$ and $i \in [1..k]$, where v_1, \dots, v_d are the same for all k nodes and x_1, \dots, x_k are distinct.
 - b. Modify T by removing the edges from the common parent of the nodes n_1, \dots, n_k and replacing them with an edge from that common parent to a new node b_j and with edges from b_j to each node n_1, \dots, n_k .
 - c. Set the value for node b_j to $\langle v_1, \dots, v_{n-1} \rangle$.
 - d. If w_1, \dots, w_k are the weights on the edges from n_1, \dots, n_k to their original parent in T , set the new weight on the branch from b_j to each n_i to $w_i/(w_1 + \dots + w_k)$ and the weight on the branch from b_j to its parent to be $w_1 + \dots + w_k$.
- iii. Set S to the (upper) subtree of T consisting of all nodes from the root up to and including the new nodes (i.e., b_1, \dots, b_m) added in step ii.

Step 4. Return T .

This algorithm is rather easier to do than to precisely describe, so try it out on some examples with the following activity.

Activity. Generate several Kinds in a mixture of unfolded and compact forms. Apply the algorithms to convert each to the other form. Use the playground to view each Kind in both forms and check your answers.

Checkpoints

After reading this section you should be able to:

- Explain what it means for two Kinds to be equivalent.
- Determine if two Kind trees are equivalent.
- Convert a Kind tree into canonical form via Algorithm CANONICAL
- Apply Algorithms COMPACT and UNFOLDED to convert back and forth between the compact and unfolded views of a Kind.

4 Building with Mixtures

Key Take Aways

A **mixture** builds a higher-dimensional FRP from two or more lower-dimensional FRPs. It selects one of several FRPs of the equal dimension (the targets) to activate *contingent* on the output value of another FRP (the mixer). We hook each of the output ports of the mixer to an input port of one of the targets, and when the mixer is activated, it triggers the rest, producing a value that concatenates the value produced by the mixer and the activated target. Mixtures capture a common feature of many random processes: contingent evolution.

An **independent mixture** is a special case where the value of the mixer does not influence the values of the target. The \star operator denotes independent mixture for FRPs and for Kinds. The independent mixture of FRPs $X \star Y$ is an FRP with a value that concatenates the values of FRPs X and Y into a single tuple.

The independent mixture of Kinds $K_1 \star K_2$ is formed by attaching a copy of K_2 at each leaf node of K_1 , concatenating the corresponding values of K_1 to the values at each node of K_2 .

The two operations are related:

$$\text{kind}(X \star Y) = \text{kind}(X) \star \text{kind}(Y). \quad (4.1)$$

An independent mixture of n FRPs of the same Kind, or of a Kind with itself n times, is denoted $x \star \star n$, an *independent mixture power*.

A function that maps a set of m -dimensional values to FRPs of dimension n is called a **conditional FRP** of *type* $m \rightarrow n$. Every FRP of dimension n is a conditional FRP of type $0 \rightarrow n$.

A function that maps a set of m -dimensional values to Kinds of dimension n is called a **conditional Kind** of *type* $m \rightarrow n$. Every Kind of dimension n is a conditional Kind of type $0 \rightarrow n$.

If r and s are compatible conditional FRPs (or Kinds) of types $m \rightarrow n$ and $n \rightarrow p$, their **mixture** $r \triangleright s$ is a conditional FRP (or Kind) of type $m \rightarrow p$.

In the mixture $R \triangleright S$ of conditional FRPs, the value produced by R 's selected

FRP is passed as input to S . And

$$\text{kind}(R \triangleright S) = \text{kind}(R) \triangleright \text{kind}(S) \quad (4.2)$$

where $\text{kind}()$ on a conditional FRP gives the corresponding conditional Kind.

In the mixture of conditional Kinds $r \triangleright s$, we copy of each Kind from s at the corresponding leaf node in r , concatenating values accordingly.

The **conditioning** operation $C \parallel K$ is a combination of the mixture $K \triangleright C$ and a projection that extracts the value of the second stage.

A common feature of random processes is contingent evolution: first something happens, then something else happens that depends on what happened initially, then something else happens that depends on what happened earlier, and so on. Mixtures capture this idea by using the value of one FRP to contingently activate other FRPs.

Figure 23 shows an example of a mixture. The FRP on the left, which we call the *mixer*, represents a random choice among three boxes corresponding to values -1 (left), 0 (center), and 1 (right). Within each box is a random amount of money, represented by the FRPs on the right, which we will call the *targets*. The targets here have different Kinds, which is fine, but we require that they have the same dimension. The mixer's $\langle -1 \rangle$ output port is connected to the top target's input port, the mixer's $\langle 0 \rangle$ output port to the middle target's input port, and mixer's the $\langle 1 \rangle$ output port to the bottom target's input port. These connections disable the mixer's display and the targets' buttons and also reconfigure the targets' displays. When the mixer is activated, the mixer output port that corresponds to the FRP's value is triggered. This value is passed to the connected target, activating it. The activated target displays both the received value from the mixer and its own value, concatenated into one tuple. The Figure shows one sample outcome of this process: the mixer produces a value of -1, which triggers the top target FRP that in turn produces a value of $\langle 2, 4 \rangle$, and the combined value $\langle -1, 2, 4 \rangle$ is displayed.

Taken together, this construction gives us what is effectively a new FRP: we push the button (on the mixer) and a value is displayed (on one of the target screens). That value, once produced, is fixed for all time. The combined value tells us what was produced by the mixer and by the *activated* target, thus reflecting a process evolving contingently.

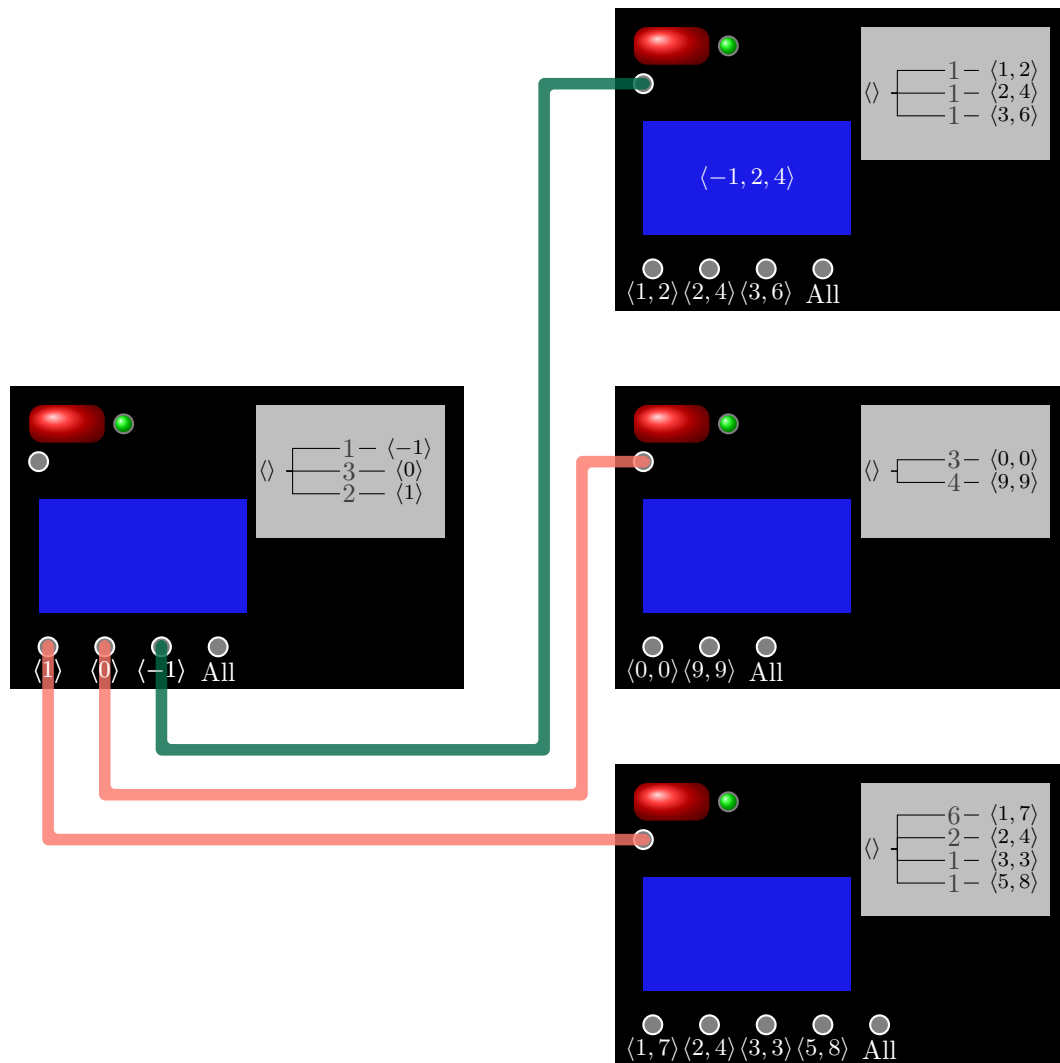
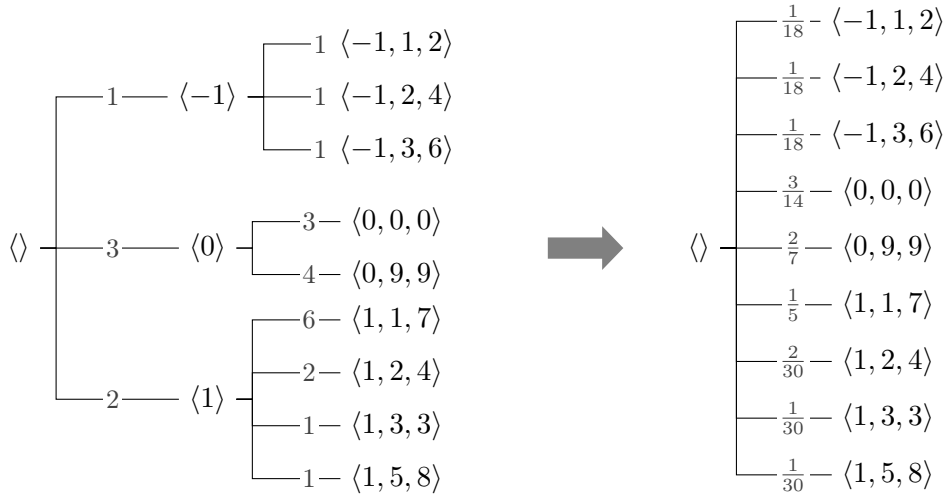


FIGURE 23. An FRP mixture, with the mixer on the left and the targets on the right. In this case, the activated mixer has value $\langle -1 \rangle$, which activates (green wire) the top target which has value $\langle 2, 4 \rangle$. The combined value $\langle -1, 2, 4 \rangle$ is displayed.

In general, a **mixture** builds a higher-dimensional FRP from several lower-dimensional FRPs. One of these is called the mixer, and the rest are called targets. The targets can have different Kinds but must all have the same dimension. The mixture FRP selects one of the targets to activate *contingent* on the output value of the mixer. We connect the mixer's output ports to the targets' input ports, with the output port for each possible value of the mixer connecting to a potentially distinct target. These connections disable the mixer's display and the targets' buttons. When the mixer is activated, its value is passed through the output port associated with that value and activates the target connected to it. The activated target displays a value that combines the mixer's value and its own.

To understand the structure of a mixture, we only need to know the Kinds of the FRPs involved and how they are connected. So if we need to illustrate mixtures, we can use a more stylized format than in Figure 23, a **wiring diagram**, with the FRPs as boxes labeled by their Kinds (as needed) and the wires emitting from the boxes ordered the same way as the leaves of the Kind. Figure 24 reproduces the mixture of Figure 23 as a wiring diagram. This mixture forms an FRP with Kind given in unfolded and canonical forms by



The mixture FRP has dimension 3 and size 9.

Figure 25 gives an even simpler example where all of the FRPs involved have the same Kind, which is shown once and indicates the order of wires for all the FRPs. This example illustrates *repeated mixtures*, where the FRP produced by the mixture

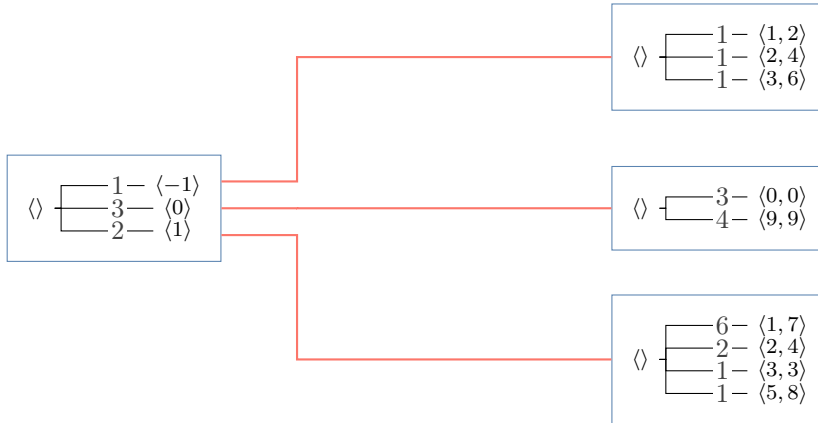


FIGURE 24. A leaner depiction of the mixture in Figure 23 as a wiring diagram. Each box is an FRP, labeled by its Kind. The wires connected to the output ports emerge from the right of the box in the same order as the Kind's leaves.

becomes the mixer for a new mixture and so on. Equivalently, each of the targets of the original mixture become the mixers for a new mixture, and in turn their targets become the mixers for yet another mixer, and so on.

Our goal in this section is to understand the mixture operation: what it means, how to use it, how to find the Kind of a mixture, and how to build mixtures in the playground. We think of a mixture as proceeding in *stages*, with the value of the mixer FRP being the initial stage, the value of the activated target FRP the second stage, the value of the FRP activated as by that target the third stage, and so on through however many mixtures we have. The simplest mixtures are those where the Kinds of the targets do not depend on the value of the mixture. These are called *independent mixtures*. We start with these.

4.1 Independent Mixtures

Think back to the Monty Hall game in Section 1.4. For any given strategy, the outcome is determined by two stages: Monty's choice of a door and your choice of a door. However, those two choices *do not interact*: you choose a door in exactly the same way whatever Monty does. You do not know what his choice was and are completely uninfluenced by it.

To reflect that with an FRP, we start with two FRPs, M (for Monty) representing Monty's choice and Y for your choice. We combine these to form a new two-

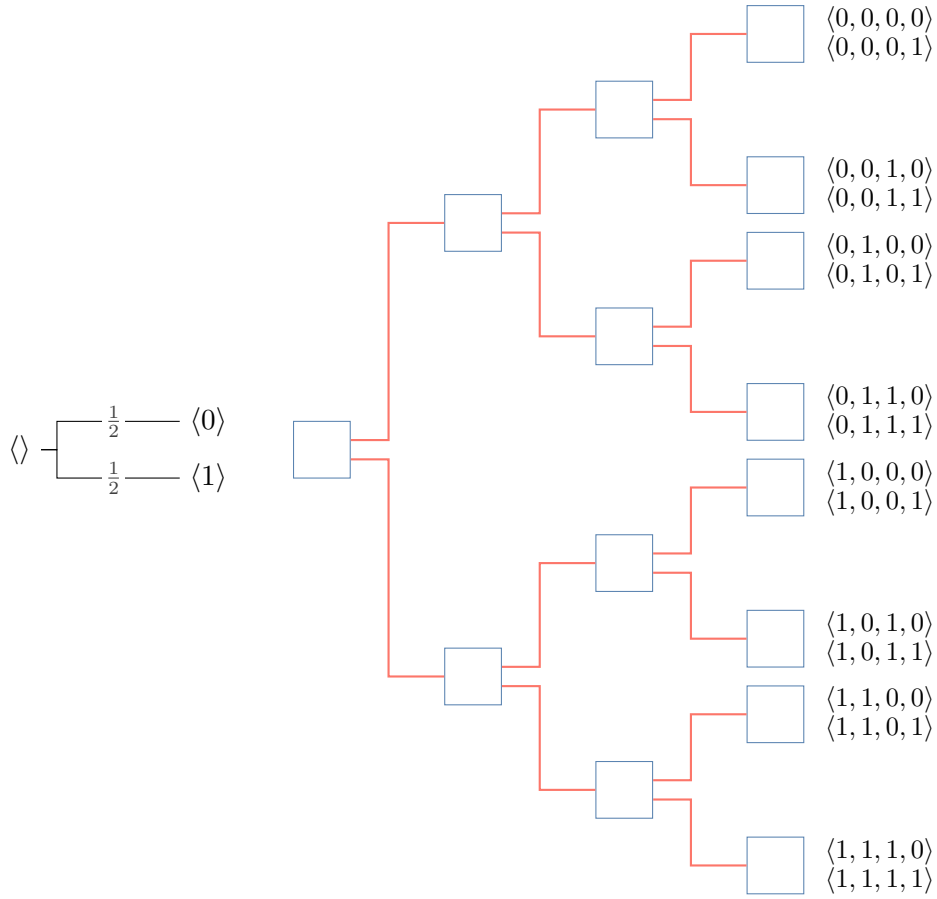


FIGURE 25. A mixture where all the FRPs (blue boxes) have the same Kind, which is shown at left. The wire connections are in the same order as the leaves of the Kind. This is a repeated mixture composed of three mixture operations. The resulting FRP has dimension 4 and size 16; its values are all 4-tuples whose components are 0 or 1. These values are listed to the right of the wiring diagram for reference, with a pair of values adjacent to the ultimate target FRP on which they will be displayed.

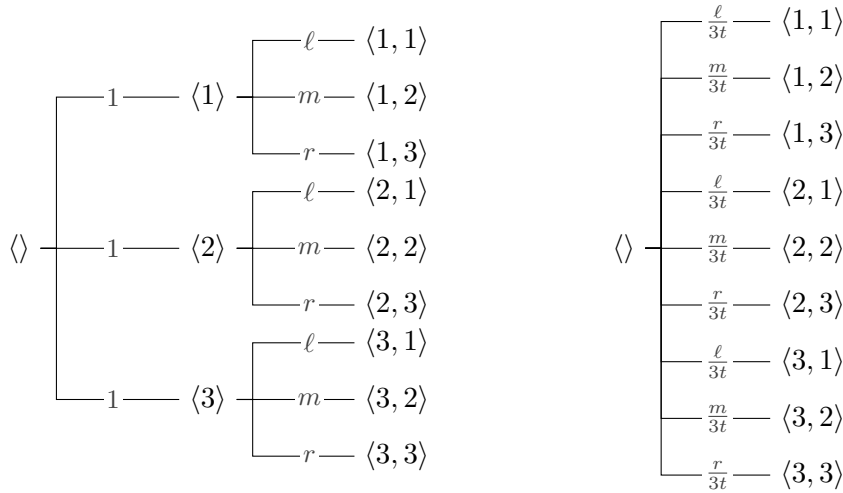
dimensional FRP whose first component is the value of M and the second component is the value of Y . This is a mixture, with wiring diagram:



Because Monty's and your choices do not interact, the behavior of Y does not depend on the value of M , so we can form this mixture by simply connecting the All output port of M to the input port of Y . This has several effects:

- Y 's button is disabled, and Y is instead activated when M produces a value.
- M 's display is disabled, and when M 's button is pushed, Y 's display shows the combined value, and
- Y 's output ports reconfigure (and relabel) automatically to give access to the combined value.

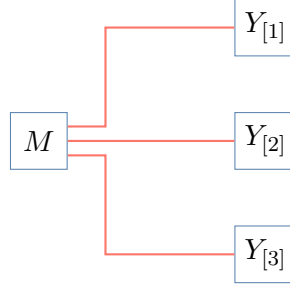
This FRP is called the **independent mixture** of M and Y ; it has Kind, in unfolded and canonical forms, with $t = \ell + m + r$,



Notice how the unfolded Kind reflects the structure of the process. Monty picks a door (stage one), then you pick a door (stage 2). But your choice does not use any

information about Monty’s choice, so the Kinds at every branch in the second level of the tree are the same.

We can form this mixture in different but equivalent way. Suppose that we have three clones of Y , i.e., FRPs with the same Kind as Y . $Y_{[1]}, Y_{[2]}, Y_{[3]}$, and we connect the $\langle i \rangle$ output port of M , for each $i \in [1..3]$, to the input port of $Y_{[i]}$. This has the wiring diagram:



While this mixture is wired differently than the earlier mixture, it behaves in the same way. As we have seen, the clones $Y_{[1]}, Y_{[2]}, Y_{[3]}$ have the same kind and are in practice interchangeable. Whichever one is activated, it will behave just like Y . Both constructions of an independent mixture are equivalent, and we can use whichever is convenient at any point. We call the former the “flat” construction, where we connect the All port of the mixer to the input port of the target, and the latter the “clone” construction, where we connect each output port of the mixer to the input port on a clone of the target.

The mixture FRP depicted in Figure 25 is also an independent mixture. The FRPs at each stage produce a value 0 or 1, regardless of what happens at any earlier stage. With four stages, we get $2^4 = 16$ possible values, each a four-dimensional tuple, as shown in the Figure. The “flat” construction for this mixture has wiring diagram



where each FRP has the same Kind as shown in Figure 25. Each FRP produces either 0 or 1 and passes that value forward in the mixture, where we get a tuple of four random 0s or 1s, each chosen independently of each other.

The independent mixture of two FRPs X and Y takes the value x produced by X and the value y produced by Y and outputs the concatenated tuple $x :: y$ joining both values. (We use $x :: y$ to denote the concatenation of tuples x and y as described

in Section F.7.) Independent mixture is a *combinator* that combines several FRPs to build a new, related FRP. We use the \star operator to denote this combinator.

If X and Y are FRPs, their **independent mixture** is the FRP denoted by $X \star Y$.

If X has dimension d_1 and size s_1 and Y has dimension d_2 and size s_2 , then $X \star Y$ has dimension $d_1 + d_2$ and size $s_1 s_2$. The values of $X \star Y$ are all the values $x :: y$ where x is a value of X and y is a value of Y .

The wiring diagram for this mixture is



though it can be written equivalently as the “clone” version.

Let us construct the mixture $M \star Y$ for the Monty Hall game in the playground. To get an FRP for Y , we need to set ℓ, m, r to specific values, here $\ell = m = r = 1$. The substitution function replaces symbolic variables with alternate values.

```

pgd> door_with_prize = uniform(1, 2, 3)
pgd> chosen_door = arbitrary(1, 2, 3, names=['l', 'm', 'r'])
pgd> M = frp(door_with_prize)
pgd> Y = frp(substitution(chosen_door, l=1, m=1, r=1))
pgd> M * Y
An FRP with value <3, 1>
pgd> M
An FRP with value <3>
pgd> Y
An FRP with value <1>
pgd> clone(M * Y)
An FRP with value <2, 2>
pgd> FRP.sample(10_000, M * Y)
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| <1, 1> | 1104 | 11.04% |
| <1, 2> | 1113 | 11.13% |

```

<1, 3>	1127	11.27%
<2, 1>	1121	11.21%
<2, 2>	1101	11.01%
<2, 3>	1088	10.88%
<3, 1>	1089	10.89%
<3, 2>	1124	11.24%
<3, 3>	1133	11.33%
+-----+-----+-----+-----+		

We create the two FRPs from their Kinds using the `frp` function. The `*` operator is the playground version of the independent mixture operator \star . The value of the mixture FRP $M * Y$ combines the values of M and Y as shown. The `clone` functions creates a clone of the given FRP, a fresh FRP with the same Kind. The sample includes all nine possible values, which occur with roughly the same frequency.

As we have seen, we can wire mixtures of FRPs over any number of stages, and indeed the independent mixture operation can be applied to any number of FRPs. For instance, we write $X \star Y \star Z$ for the three stage mixture of the FRPs X, Y, Z . In general, $X_1 \star X_2 \star \dots \star X_n$ is an independent mixture of n stages with wiring diagram



Example 4.1. In the classic game *Dungeons & Dragons*, each player has a character with six attributes (Strength, Intelligence, Wisdom, Constitution, Dexterity, Charisma) determined by an integer score in $[3..18]$. Each attribute's score is determined by rolling three six-side dice and summing their values.

If D_1, D_2, D_3 are FRPs each representing a roll of a balanced six-sided die, the independent mixture $D_1 \star D_2 \star D_3$ represents the three rolls. Using an independent mixture means that the value of any individual roll does not influence the value any other roll. If we observed D_1 's value, say, it would not help us predict the value of D_3 . Then, the FRP $\text{Sum}(D_1 \star D_2 \star D_3)$ represents the value of an attribute.

If S, I, W, C_o, D, C_h are FRPs that represent the six attributes' scores, the independent mixture $S \star I \star W \star C_o \star D \star C_h$ represents a character. Using an independent mixture again tells us that the processes generating the different

attribute scores do not interact. Whatever score we roll for Strength, for instance, does not influence (or help us predict) the score for Wisdom or Dexterity.

We can generate these in the playground. Rather than do this over and over, we will write FRP factories to create fresh FRPs on demand. These are just Python functions, which we can either enter directly in the playground or write in a separate Python source file.

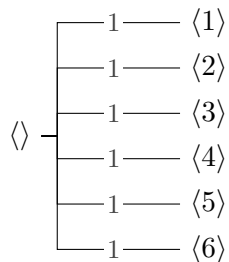
```
def dice_roll():
    "Returns an FRP representing the roll of a balanced 6-sided die."
    return frp(uniform(1, 2, ..., 6))

def dnd_attribute():
    "Returns an FRP representing a score for a D&D character attribute."
    D_1, D_2, D_3 = dice_roll(), dice_roll(), dice_roll()
    return Sum(D_1 * D_2 * D_3)

def dnd_character():
    "Returns an FRP representing a D&D character's attribute scores."
    S = dnd_attribute()    # Strength
    I = dnd_attribute()    # Intelligence
    W = dnd_attribute()    # Wisdom
    Co = dnd_attribute()   # Constitution
    D = dnd_attribute()    # Dexterity
    Ch = dnd_attribute()   # Charisma

    return S * I * W * Co * D * Ch
```

The `dice_roll` factory uses the *Kind* factory `uniform` to produce the *Kind* of a single roll and `frp` to create a fresh FRP with that *Kind*. Note that the `...` in the call to `uniform` is intentional. It extends the pattern of the first two values up to and including the value after the `...`, so `uniform(1, 2, ..., 6)` is the *Kind*



As above, both `dnd_attribute` and `dnd_character` use independent mixtures, the former of dimension 3 and the latter of dimension 6.

We can use these in the playground. If we typed the definitions in at the prompt, we can refer to them directly. If we entered the definition in a file `dnd.py`, say, then we first type `from dnd import dnd_attribute, dnd_character` at the playground prompt.

```
pgd> dnd_attribute()
An FRP with value <17>
pgd> dnd_character()
An FRP with value <13, 8, 9, 11, 8, 14>.
pgd> dnd_character()
An FRP with value <14, 7, 8, 11, 14, 16>.
```

If desired, we could easily define statistics that extract character's attribute scores by name.

```
pgd> Strength = Proj[1]
pgd> Intelligence = Proj[2]
pgd> Wisdom = Proj[3]
pgd> Constitution = Proj[4]
pgd> Dexterity = Proj[5]
pgd> Charisma = Proj[6]
pgd> char1 = dnd_character()
pgd> char2 = dnd_character()
pgd> char3 = dnd_character()
pgd> Strength(char1)
```

```

An FRP with value <9>
pgd> Wisdom(char2)
An FRP with value <16>
pgd> char3 ~ Fork(Intelligence, Charisma)
An FRP with value <14, 17>

```

Multi-stage mixtures like $D_1 \star D_2 \star D_3$ or $S \star I \star W \star D \star C_h \star C_o$, are well defined because the \star operator is *associative*.⁴⁰ This means that for any FRPs X, Y, Z , the mixtures $(X \star Y) \star Z$ and $X \star (Y \star Z)$ are the same FRPs. Any way of grouping mixtures in a multi-stage mixture thus gives the same result.

⁴⁰ Associativity and commutativity of operations is discussed in detail in Section F.9.

We have created mixture FRPs by wiring together various other FRPs. This produces a device that acts and quacks just like an FRP even if it is not in one box, so we treat it as one. If, however, we could find the *Kind* of that FRP, we could simply order an FRP with that Kind from the Warehouse and have our mixture in a nice clean package. In the Monty Hall game discussed earlier, $\text{kind}(M \star Y)$ is given on page 128. Where did this come from? And how does it relate to $\text{kind}(M)$ and $\text{kind}(Y)$? This leads to a key question: **can we find the Kind of an independent mixture from the Kinds of the constituent FRPs?**

The answer is yes. We will define an independent-mixture operation on *Kinds*, denoted with the same operator \star , that makes the following identity hold:

$$\text{kind}(M \star Y) = \text{kind}(M) \star \text{kind}(Y). \quad (4.3)$$

In words: the Kind of an independent mixture FRP is the independent mixture of the two Kinds.

The \star operation on Kinds directly follows the wiring diagram for the “clone” construction in two steps:

1. attach a copy of the $\text{kind}(Y)$ tree to each leaf node in $\text{kind}(M)$
2. rewrite the new leaf nodes to hold the concatenated tuples of values seen on the path from root to leaf.

Figure 26 illustrates this. On the left side of the figure, we associate a copy of $\text{kind}(Y)$ at each leaf node of $\text{kind}(M)$. We join those trees, giving new leaf nodes for each copy of $\text{kind}(Y)$. We then rewrite the values of those leaf nodes to hold the sequence of values seen as we move from the root to that leaf. For example, top-most node is on the $\langle 1 \rangle$ branch for M and the $\langle 1 \rangle$ branch for Y ; its value becomes $\langle 1, 1 \rangle$.

The fourth leaf node down is on the $\langle 2 \rangle$ branch for M and the $\langle 1 \rangle$ branch for Y ; its value becomes $\langle 2, 1 \rangle$. The eighth leaf node down is on the $\langle 3 \rangle$ branch for M and the $\langle 2 \rangle$ branch for Y ; its value becomes $\langle 3, 2 \rangle$. And so on.

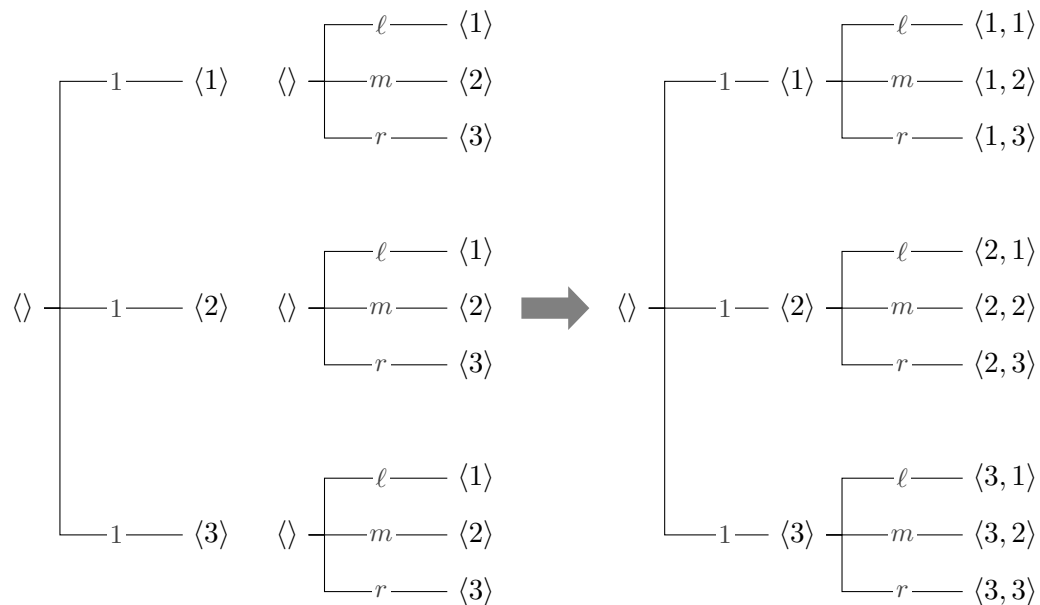


FIGURE 26. Constructing the Kind mixture $\text{kind}(M) \star \text{kind}(Y)$. For each leaf of $\text{kind}(M)$, we take a *copy* of $\text{kind}(Y)$ and attach it, forming the Kind tree on the right.

Puzzle 25. Convince yourself that with this definition of independent mixture of Kinds, $\text{kind}(M \star Y) = \text{kind}(M) \star \text{kind}(Y)$. Notice that after M generates a value, the next stage looks the same no matter what that value is.

We can examine these Kinds in the playground. Using the Kinds `door_with_prize` and `chosen_door` defined earlier (and in `frplib.examples.monty_hall`) and the FRPs M and Y , do

```
pgd> door_with_prize
pgd> chosen_door
pgd> outcome = door_with_prize * chosen_door
pgd> outcome
```

```

pgd> outcome1 = substitution(outcome, l=1, m=1, r=1)
pgd> outcome1
pgd> kind(M * Y)
pgd> Kind.equal(outcome1, kind(M * Y))

pgd> unfold(substitution(door_with_prize * chosen_door, l=1, m=1, r=1))
pgd> unfold(kind(M * Y))

```

The output is omitted here, but it should show the corresponding Kinds equal and match the Figures up to the scaling of the weights.

If K_1 and K_2 are Kinds, their \star is a Kind $K_1 \star K_2$, defined by the procedure:

1. attach a copy of K_2 to each leaf node of K_1 , and
2. replace each leaf node of the combined tree from step 1 with a concatenation of the tuples on the path from the root to the leaf.

This operation satisfies a key identity: if X and Y are FRPs, then

$$\text{kind}(X \star Y) = \text{kind}(X) \star \text{kind}(Y), \quad (4.4)$$

so Kinds and FRPs combine in similar ways.

As with FRPs, we take independent mixtures of Kinds over any number of stages, e.g., $K_1 \star K_2 \star \cdots \star K_n$. Again \star is an associative operation. And equation (4.4) tells us that for FRPs X_1, X_2, \dots, X_n we have

$$\text{kind}(X_1 \star X_2 \star \cdots \star X_n) = \text{kind}(X_1) \star \text{kind}(X_2) \star \cdots \star \text{kind}(X_n). \quad (4.5)$$

The Kinds of independent mixtures are thus completely determined by the Kinds of their constituent FRPs.

Example 4.2. In a round-robin tournament, twelve players are ranked from 1 to 12. The first match pits two players, with one chosen at random from the top six ranks and the other chosen at random from the bottom six ranks. The two choices are made independently. Assume that players are weighted by rank in this choice, as shown in Figure 27 for the first player chosen ($s = 0$) and the

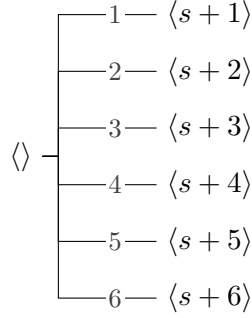


FIGURE 27. The Kind for the choice of player matchups in Example 4.2, with $s = 0$ for the first player and $s = 6$ for the second player.

second player chosen ($s = 6$). Let F represent the pair of players engaged in the first match. We want to find $\text{kind}(F)$.

Let `first_player` and `second_player` be the Kinds for the first and second player chosen, from Figure 27.

```
pgd> first_player = weighted_by(1, 2, ..., 6, weight_by=scalar_fn(Id))
pgd> second_player = weighted_by(7, 8, ..., 12, weight_by=scalar_fn(Id - 6))
pgd> first_player
pgd> second_player
```

The Kind factory `weighted_by` returns a Kind with the given values where the weights are computed by a function of the value. (Here, `scalar_fn` converts a scalar statistic into a simple function.) Look at these Kinds and compare to the display above.

We have that $F = \text{frp}(\text{first_player} * \text{second_player})$ and $\text{kind}(F)$ equals `first_player * second_player`. Before we compute this Kind in the playground, it is worth seeing how we would compute it by hand.

Let's consider the leaf node of the independent mixture Kind with value $\langle 3, 10 \rangle$. The $\langle 3 \rangle$ node in `first_player` has weight $\frac{3}{21}$, and the $\langle 10 \rangle$ node in `second_player` has weight $\frac{4}{21}$. So the canonical weight on node $\langle 3, 10 \rangle$ is $\frac{4}{147} \approx 0.027211$. Similarly, for the leaf node $\langle 6, 12 \rangle$, the `first_player` weight for $\langle 6 \rangle$ is $\frac{6}{21}$ as is the `second_player` weight for $\langle 12 \rangle$. The weight on the leaf node $\langle 6, 12 \rangle$ is $\frac{4}{49} \approx 0.081633$.

With this in mind, we can look at the Kind, in both unfolded and canonical

form. The output is omitted here, but see Figure 28 for the unfolded form.

```
pgd> match_up = first_player * second_player
pgd> unfold(match_up)
pgd> match_up
```

Notice that

```
pgd> Kind.equal(Proj[1](match_up), first_player)
True
pgd> Kind.equal(Proj[2](match_up), second_player)
True
```

Puzzle 26. Convince yourself that the “flat” and “clone” constructions of $M \star Y$ give FRPs with the same Kind.

Puzzle 27. In Example 4.1, sketch the Kind of

```
dice_roll() * dice_roll() * dice_roll()
```

by hand. How would you find the Kind of `dnd_attribute()` by hand?

Find both these Kinds in the playground to check your work.

The word **independent** in “independent mixture” has meaning. It tells us that the distinct stages of the process represented by the FRPs in the mixture evolve without interaction or influence on each other. So if you observe the value of some FRPs in the mixture, that information *does not help you predict the value of any other FRPs* in the mixture. It is independence that leads to equations (4.4) and (4.5). It is independence that gives the “flat” and “clone” constructions equivalent outputs. When the parts of a system are independent, we can analyze the whole system by analyzing the parts separately. Lack of independence – *dependence* – means that knowing the values of some FRPs in the mixture *changes our predictions* of the other FRPs’ values. A system with dependent parts is coupled and cannot be as easily decomposed. Independence, when we have it, is powerful.

Figure 28 from the previous example illustrates these ideas. Without any information, what can we say about the second player in the match-up? Our knowledge

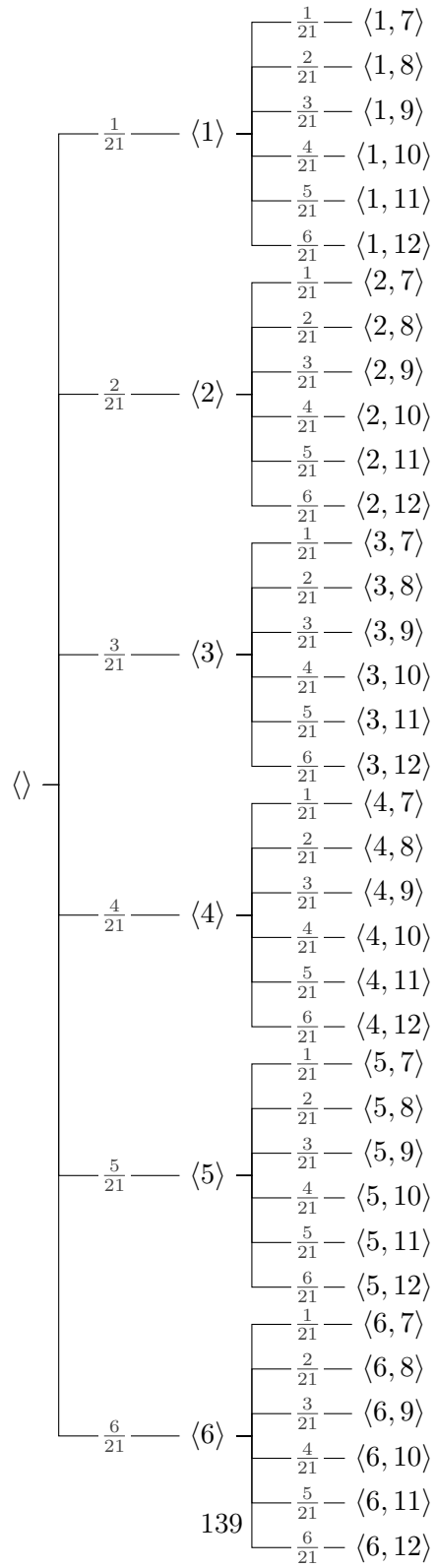
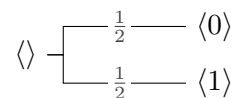


FIGURE 28. Unfolded Kind of the first tournament matchup in Example 4.2.

(and all the predictions we might make about the second plays) is embodied in the Kind `second_player`, which is just `Proj[2](match_up)`. Now suppose I tell you (truthfully) that I have activated and observed the display of the first player rank FRP and its value is $\langle 4 \rangle$. How does this change your knowledge of the second player? Picture yourself at the $\langle 4 \rangle$ node in the Kind tree, having just obtained the information about the first player. What will happen in the next stage is represented by the subtree at that node. But that is exactly The *definition* of the independent mixture of Kinds means that for any value $\langle v \rangle$ of the first player FRP, the Kind we see *looking down the tree from the $\langle v \rangle$ node* is just a copy of `second_player`. The knowledge of the first player gives us no useful information about the second player. Similarly, if I observe that the second player is ranked 9 and tell you this. You know that you must be at one of the leaf nodes that look like $\langle \blacksquare, 9 \rangle$; you do not know which one. Looking up the tree The knowledge of the second player gives us no useful information about the first player. The FRPs representing choices of the two players are independent.

The choice to use an independent mixture when building a system is an assumption that the constituent FRPs are independent in this sense. The next example shows this idea as well, and we will return formally to the idea of independence when we discuss conditionals and predictions in more depth. General mixtures (subsection 4.3) introduce dependence into the mix (so to speak).

Example 4.3. We want to build an FRP representing three flips of a coin where heads and tails are equally likely and each flip is independent of the other flips. The result will be an independent mixture $F = F_1 \star F_2 \star F_3$ where the FRPs F_i all have Kind



where we use 0 to represent tails and 1 to represent heads.

In the playground, this becomes

```
pgd> flip = either(0, 1)
pgd> F_1, F_2, F_3 = frp(flip), frp(flip), frp(flip)
pgd> F = F_1 * F_2 * F_3
```

```

pgd> F
An FRP with value <1, 0, 1>
pgd> flips = flip * flip * flip
pgd> Kind.equal( kind(F), flips )
True

```

Look at the Kind `flips` in the playground. (Really!) Imagine that you have learned the value of F_1 and want to assess what this information tells you about F_2 and F_3 . If move in the kid tree to the node representing the observed value of F_1 , what you see looking down the tree is the Kind of the other two flips, but for either value of F_1 , you see the same tree: `flip * flip`. Your predictions about F_2 and F_3 have not changed. If instead you have learned the value v of F_3 , then you know that you are at a node for which F_3 takes that value (i.e., $\langle \blacksquare, \blacksquare, v \rangle$), but you cannot tell which one. Looking up from all such nodes gives you a tree that again is just `flip * flip`, and again your predictions of the other flips have not changed. Although it looks more complicated, the same story holds if you learn the value of F_2 . This puts you at a node like $\langle \blacksquare, v, \blacksquare \rangle$ and what you can see from all such nodes is again just the tree `flip * flip`.

Another way to see the latter is to transform the Kind with a permutation so that the third flip is listed first:

```
pgd> flips_p = flips ^ Permute(2,1)
```

Now, if we have observed F_3 , our analysis is like that of F_1 earlier, looking down the tree from the nodes at the first level. In fact, for any permutation of the three components, the Kind transformed by the permutation is equal to `flips`.

The above analysis works with various Kinds of information. For instance, observing the values of F_1 and F_2 , puts us at a node in the second level, and looking down the tree, we see `flip`. Conditionals (Section 5) express this idea.

Puzzle 28. If K_1 and K_2 are Kinds, is $K_1 \star K_2$ the same as $K_2 \star K_1$? If so, how do you know? If not, how are they related?

Aside: The Algebra of Independent Mixtures. The independent mixture operation \star has a few notable properties. First, let X be an FRP. Recall that **empty** is the trivial FRP with Kind $\langle \rangle$, just a root node with an empty list.

If we connect the All output port of **empty** to the input port of X , then when we push the button on **empty**, we get just the output of X because the empty list does not add anything to the value. So, $\text{empty} \star X$ is the same as X .

Similarly if we connect the All output port of X to **empty** (or the individual output ports of $\text{size}(X)$ copies of **empty**), the result is again the same as X . So, $X \star \text{empty}$ is the same as X . That is:

$$X \star \text{empty} = X = \text{empty} \star X.$$

This applies to the Kinds as well:

$$\text{kind}(X) \star \langle \rangle = \text{kind}(X) = \langle \rangle \star \text{kind}(X),$$

where $\langle \rangle$ denotes the empty Kind ($\text{kind}(\text{empty})$). For FRPs and Kinds, respectively, **empty** and $\langle \rangle$ are “identity elements” for the independent mixture operation.

Second, with three FRPs X , Y , and Z , we can take the independent mixture of the three in two different ways: $X \star (Y \star Z)$ or $(X \star Y) \star Z$. As we are just connecting output and input ports, it does not matter which pair we mix first, and similarly with Kinds:

$$\begin{aligned} X \star (Y \star Z) &= (X \star Y) \star Z \\ \text{kind}(X) \star (\text{kind}(Y) \star \text{kind}(Z)) &= (\text{kind}(X) \star \text{kind}(Y)) \star \text{kind}(Z). \end{aligned}$$

Thus, \star is “associative” for FRPs and Kinds, and we can write the mixture without parentheses, $X \star Y \star Z$ and $\text{kind}(X) \star \text{kind}(Y) \star \text{kind}(Z)$.

A set of objects (here either FRPs or Kinds) with an associative binary operation and an identity element is called a **monoid**. See Section F.9.1 for much more on this important and ubiquitous algebraic idea.

Note that the \star operator is not *quite* commutative, but it is close in an important sense. $X \star Y$ and $Y \star X$ are not equal in general because they combine

their constituent values in different orders within the value, and similarly for Kinds $K_1 \star K_2$ and $K_2 \star K_1$. However, we can transform $X \star Y$ to $Y \star X$ and vice versa by permuting the tuples, so they have effectively the same information. The same goes for Kinds $K_1 \star K_2$ and $K_2 \star K_1$. Formally, this transform is an invertible permutation statistic ψ such that $Y \star X = \psi(X \star Y)$ and $X \star Y = \psi^{-1}(Y \star X)$ and $K_2 \star K_1 = \psi(K_1 \star K_2)$ and $K_1 \star K_2 = \psi^{-1}(K_2 \star K_1)$. Thus, the two FRPs $X \star Y$ and $Y \star X$ and the two Kinds $K_1 \star K_2$ and $K_2 \star K_1$ are the same up to ordering of the components. While not *equal*, we say that they are *isomorphic*, which is the next best thing.

See Sections [F.5](#) and [F.6](#) for details on invertible functions.

A common pattern is to build an independent mixture several of an FRP or Kind with itself some number of times, like

```
dice_roll() * dice_roll() * dice_roll()
```

This looks and acts like a “power”, and because this is so common, we have a shorthand for it that evokes that idea.

For any Kind k and any natural number m , we define

$$k \star\star m = \overbrace{k \star \cdots \star k}^{m \text{ times}}, \quad (4.6)$$

where $k \star\star 0 = \langle \rangle$, the empty Kind.

For any FRP X and any natural number m , we define

$$X \star\star m = \overbrace{\text{clone}(X) \star \text{clone}(X) \star \cdots \star \text{clone}(X)}^{m \text{ times}}, \quad (4.7)$$

where $X \star\star 0 = \text{empty}$. The clones are here to make the shorthand more useful; simply repeating the exact value of X is not what we usually want. This ensures that $X \star\star m$ is an independent mixture of m FRPs with the same Kind as X .

In the playground, we use the `**` operator for this, so these mixtures look like `X ** m` and `k ** m`.

The operators `⋆⋆` and `**` are reminiscent of the operator for powers in several

programming language, which is not a coincidence. Indeed, $k \star \star m$ and $X \star \star m$ are essentially *powers* of the independent mixture operator.

Example 4.4 Gambler's Fortune

A gambler places a series of identical bets of \$1 on a casino game, e.g., roulette (Example 2.1). Let B_1, B_2, \dots, B_n be the FRPs representing the outcomes of individual bets. We assume that these all have Kind

$$\langle \rangle \begin{cases} 1-p & \langle -1 \rangle \\ p & \langle 1 \rangle \end{cases}$$

We can get this Kind in the playground in several ways, though we need to set a value for p (here 4/9) to generate concrete FRPs.

```
pgd> p = symbol('p')
pgd> bet_kind = weighted_as(-1, 1, weights=[1 - p, p])
pgd> bet_kind_c = substitution(bet_kind, p=as_quantity('4/9'))
pgd> bet_kind ** 3
,---- 1 + -3 p + 3 p^2 + -1 p^3 ---- <-1,-1,-1>
|---- p + -2 p^2 + p^3 ----- <-1,-1, 1>
|---- p + -2 p^2 + p^3 ----- <-1, 1,-1>
|---- p^2 + -1 p^3 ----- <-1, 1, 1>
<> -|
|---- p + -2 p^2 + p^3 ----- <1,-1,-1>
|---- p^2 + -1 p^3 ----- <1,-1, 1>
|---- p^2 + -1 p^3 ----- <1, 1,-1>
`---- p^3 ----- <1, 1, 1>
pgd> bet_kind_c ** 3
,---- 0.17147 ----- <-1,-1,-1>
|---- 0.13717 ----- <-1,-1, 1>
|---- 0.13717 ----- <-1, 1,-1>
|---- 0.10974 ----- <-1, 1, 1>
<> -|
|---- 0.13717 ----- <1,-1,-1>
```

```

      |---- 0.10974 ----- <1,-1, 1>
      |---- 0.10974 ----- <1, 1,-1>
      `---- 0.087791 ----- <1, 1, 1>
pgd> B_c = frp(bet_kind_c)
An FRP with value <1>
pgd> B_c ** 10
An FRP with value <1, 1, -1, -1, -1, -1, -1, -1, -1, -1>

```

Here, `bet_kind ** 3` is the Kind for a series of three bets with general p , and `B_c ** 10` is an FRP representing a series of 10 bets with $p = 4/9$. Notice that `bet_kind ** n` has size 2^n which grows quickly. Look at the Kind for `bet_kind ** 5` up to, say, `bet_kind ** 10`; beyond that, the Kind tree gets rather overwhelming and soon slow to compute.

However, as usual, we are in practice less interested in the sequence of bets itself and more interested in statistics that answer questions about that sequence. For instance, what is the gambler's net gain or loss? Or: does the gambler end up ahead or behind?

```

pgd> net_gain = Sum
pgd> end_ahead = (Sum >= 0)
pgd> net_gain(B_c ** 10)
An FRP with value <-6>
pgd> end_ahead(bet_kind_c ** 5)
      ,---- 0.60331 ----- 0
      <> -|
      `---- 0.39669 ----- 1

```

For large n , these computations will get slow because $B_c \star n$ has size exponential in n . But there is a tool in the playground for computing statistics on these mixture powers efficiently. (Details are in Section 6.1, but we do not care about those details here.) With this in hand, we are in a position to answer some interesting questions. If you could choose n , what should you choose? What do we predict for your winnings and for whether you will at least break even?

So, suppose we have a $\$w$ in our wallet and will place n equal-sized bets of

$\$w/n$, on a game like the above. First, we will write a function `winnings` to do our analysis. It will take an FRP (like `B_c`) that represents a single \$1 bet along with n and w and *return the Kind of the FRP that represents our total winnings*. The statistic `in_the_black` takes the winnings and indicates whether we at least broke even, which we document below.

```
def winnings(Bet, n, w=1):
    """Returns the kind of net winnings after n bets of $w/n.

    Bet is an FRP representing a $1 bet on the game.

    """

    stakes = w / n
    bet = kind(Bet ^ (__ * stakes)) # Same game, scaled by stakes
    return fast_mixture_pow(Sum, bet, n)

in_the_black = (__ >= 0)
in_the_black.__doc__ = 'Given our winnings, have we at least broken even?'
```

Now, we can look at some results assuming an initial wealth of \$4200. We loop over selected values of n , compute the two Kinds, and store them. This may take a moment to run, though we could make this much faster with a little more effort.

```
pgd> win_kind = {} # Dictionary keyed by n
pgd> for n in [*range(1, 11), *range(15,50,5), 50, 75, 100, 500]:
    win_kind[n] = winnings(B_c, n, 4200)
pgd> in_the_black(win_kind[1])
    ,---- 5/9 ---- 0
    <> -|
    `---- 4/9 ---- 1
pgd> in_the_black(win_kind[2])
    ,---- 0.30864 ---- 0
    <> -|
```

```

    `---- 0.69136 ---- 1
pgd> in_the_black(win_kind[10])
    ,---- 0.51886 ---- 0
    <> -|
    `---- 0.48114 ---- 1
pgd> in_the_black(win_kind[100])
    ,---- 0.84550 ---- 0
    <> -|
    `---- 0.15450 ---- 1
pgd> in_the_black(win_kind[500])
    ,---- 0.99283 ----- 0
    <> -|
    `---- 0.0071681 ---- 1

```

We expect there to be a difference between even and odd n here because with even n , there is an extra way to break even. We collect these results using `E` to compute our predictions about winnings and breaking even:

```

pgd> win_pred = { n : as_float(E(k)) for n, k in win_kind.items() }
pgd> break_even = { n : as_float(E(in_the_black(k))) for n, k in win_kind.items() }
pgd> break_even[0] = 1

```

Looking at `win_pred`, which gives our predicted winnings for each n , we see that all the values are the same $-466\frac{2}{3}$. This happens to be $4200 \cdot -\frac{1}{9}$. The game B_c is after all against us in that we are more likely to lose than win, and note that $E(B_c)$ is $-1/9$ as you can confirm in the playground. Our best prediction is that we will lose on average $\$-\frac{1}{9} \cdot \frac{w}{n}$ per play.

On the other hand, `break_even` shows a pattern plotted in Figure 29. Even and odd n are indeed different as predicted, and within each group, our chance of breaking even is *strictly decreasing*. (Notice that the Kind

```
in_the_black(winnings(B, n, w))
```

does not depend on w at all.) In an “unfair” game like B_c , our best choice is

not to play, and if we play, we minimize our chance of an overall loss by “bold” play – putting it all on a few bets.

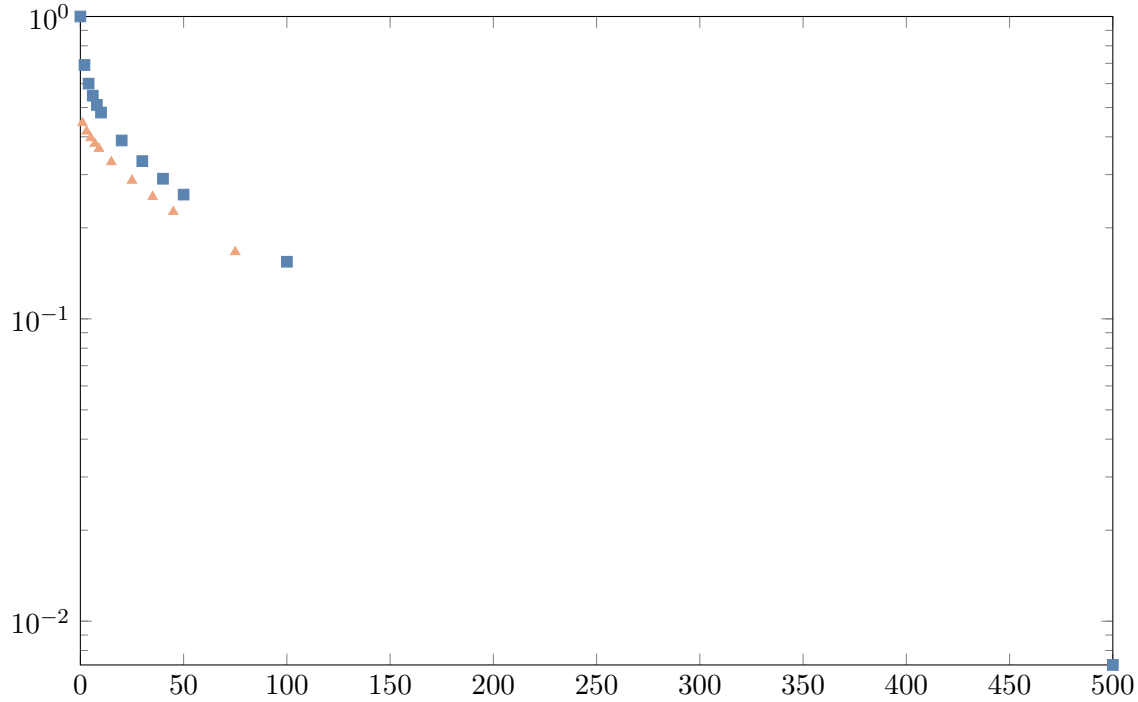


FIGURE 29. Predicted chance to break even in n bets in Example 4.4. Results for even n are plotted as squares and for odd n triangles.

4.2 Conditional FRPs and Conditional Kinds

The “clone” construction for an independent mixture suggests an immediate generalization. Instead of connecting *clones* of the target to the mixer, we can connect *different* FRPs (of the same dimension). This creates an FRP where the value produced at the second stage is *contingent* on the value produced at the first stage. This is a general **mixture**.

To formally define general mixtures, it will be useful to expand our toolbox with some new hardware. Figure 30 shows an example of a *selector switch*. On the input side, the switch has a single port that can accept values from an FRP. On the output side, the switch has a port for each of several *specific* values, which can be arbitrary

tuples of the *same dimension*. When a signal is passed through the input port, the switch checks which value that signal represents and passes the signal through the corresponding output port. An input signal must represent one of the possible output values. (If not, the input port glows red to indicate the error. Overheating or other bad things may occur.)

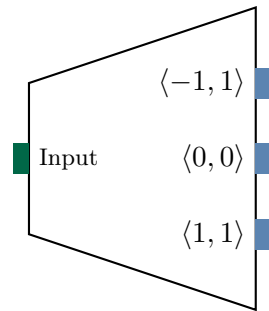


FIGURE 30. An example selector switch with input port on the left and labeled output ports on the right. This switch accepts only the listed values at its input.

Let us look at a general mixture and see how we might use a selector switch.

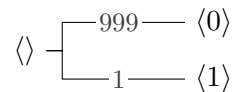
Example 4.5.

One out of a thousand people have a particular disease. When 1000 people *with* the disease are tested, roughly 950 will test positive. When 1000 people *without* the disease are tested, roughly 10 will test positive. We want to (eventually) make a good prediction on whether a patient has the disease given that they test positive. Here, we will just build an FRP to describe this system.

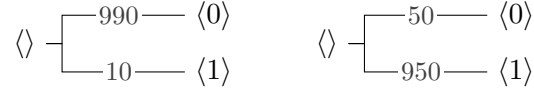
The system evolves in two stages: determine whether the patient has the disease and then, contingent on that outcome, determine the result of the test. As before, we associate a number with each outcome:

$0 \leftrightarrow$ No Disease	$0 \leftrightarrow$ Test Negative
$1 \leftrightarrow$ Disease	$1 \leftrightarrow$ Test Positive

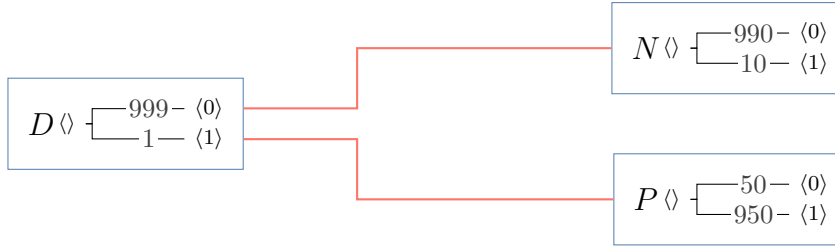
The first stage is represented by an FRP D with Kind



The second stage is represented by two FRPs, call them N and P for “negative” and “positive,” with Kinds:

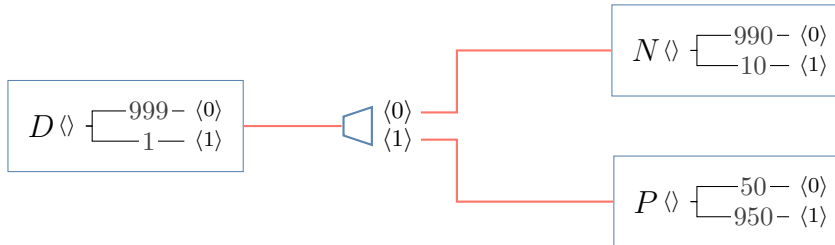


The system is described by a mixture with the following wiring diagram:



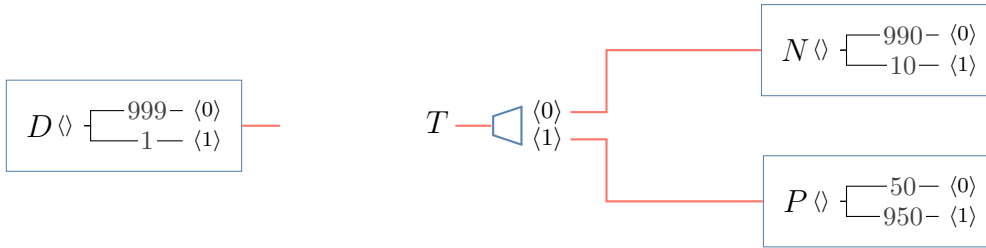
As with the independent mixtures earlier, the value produced by D determines which of N and P is activated. The only difference from those earlier cases is that N and P have different Kinds. When we push the button on D , it activates one of N and P , and the mixture shows the *combined* value on the display of the activated FRP: a tuple that concatenates the value of D and the value of the activated target (N or P).

Now, let's make one change by inserting a selector switch:



Here, we have connected D 's All output port to the input of the switch and the switch's output ports to each of the target FRPs. Despite the addition of the switch, the two wirings are completely equivalent. The switch plays the same role as the internal circuitry in the FRP that controls its output ports. We have simply reproduced that function in the switch.

What we get out of the second wiring is the ability to decompose the system differently, as in



On the left, we have D , the mixer, as before. On the right, we have an object, which we have named T , of a new type that associates each input in some pre-specified set with an FRP. When connected to a mixer FRP producing only valid inputs, like D , it acts as if we had wired the target FRPs directly to the mixer, as in the previous two diagrams. We call T a **conditional FRP**. It represents the patient's test outcome contingent on the patient's disease status.

The conditional FRP T acts like N with 0 prepended to the output when $\langle 0 \rangle$ is input and like P with 1 prepended to the output when $\langle 1 \rangle$ is input. Connecting the All output port of any FRP with values $\{\langle 0 \rangle, \langle 1 \rangle\}$ to the conditional FRP's input port gives a valid mixture FRP.

The conditional FRP in the previous example is of the DIY⁴¹ variety. We built it from existing FRPs N and P , wires, and a selector switch tailored to the chosen input values. And that's fine. Fortunately, conditional FRPs have proved so useful in practice, that the FRP Warehouse offers conditional FRPs *integrated into a single device that looks like an ordinary FRP*. It receives its input value through its input port, which is labeled with the valid input values, and its button is disabled until an input value is given. Indeed, we can see that an ordinary FRP is just a conditional FRP that takes no input, or to put it another way, the 0-dimensional input $\langle \rangle$.

⁴¹"Do it yourself"

Before looking at conditional FRPs more formally, let us play with Example 4.5 in the playground. Here, `either(u, v, weight_u)` is a Kind factory producing Kinds with two values `u` and `v` and respective weights `weight_u` and 1.

```
pgd> D = frp(either(0, 1, 999))
pgd> N = frp(either(0, 1, 99))
pgd> P = frp(either(0, 1, 1/19))
```

In the playground, we can define a conditional FRP in several ways: as a dictionary mapping input values⁴² to Kinds,

```
pgd> T = conditional_frp({ 0: N, 1: P })
```

as a named function,

```
pgd> @conditional_frp(codim=1, target_dim=1)
...> def T(value):
...>     if value == 0: # Inputs are scalars when codim=1
...>         return N
...>     return P
```

or as an anonymous function⁴³

```
pgd> T = conditional_frp(lambda value: N if value == 0 else P, codim=1)
```

`conditional_frp` used as a function or a decorator wraps the given mapping into a conditional FRP object. When the codimension is 1, a wrapped function must take a *scalar* argument; in all other cases, the function's argument will be a tuple. When the codimension is 1 and passing a dictionary, you may use scalar keys.

`conditional_frp` can deduce the valid inputs of a conditional FRP from a dictionary but not from a function. You can specify such properties of the conditional FRP with optional arguments. It is good practice to give the codimension (via `codim`) and dimension (via `dim` or `target_dim`).⁴⁴ where possible and appropriate. You can also use `domain` to specify the valid inputs, which causes an error to be raised for an invalid value. The domain can be an explicit list⁴⁵ of valid values, of common dimension, or a predicate that returns true for valid values.⁴⁶ For instance, to indicate that the function `T` above accepts only values 0 and 1:

```
pgd> @conditional_frp(domain=[0, 1], target_dim=1)
...> def T(value):
...>     if value == 0:
...>         return N
...>     return P
```

With an explicit `domain`, `conditional_frp` can deduce the codimension for a function, so it need not be supplied. Alternatively, you can handle invalid inputs in the function:

⁴²In the scalar (`codim=1`) case, the dictionary keys can be scalars or 1-dimensional tuples, written `(0,)` in Python.

⁴³In Python, anonymous functions are written with the `lambda` keyword as `lambda args: expr` taking arguments `args` and returning the value of expression `expr`.

⁴⁴The `dim` includes the length of the input vector; `target_dim` does not.

⁴⁵...or a set or a generator/iterator

⁴⁶When `codim=1`, a domain predicate should accept scalar inputs.

```

pgd> @conditional_frp(codim=1)
...> def T(value):
...>     if value == 0:
...>         return N
...>     if value == 1:
...>         return P
...>     raise MismatchedDomain(f'Value {value} for T must be 0 or 1')

```

When we examine the conditional FRP T in the playground, it shows us the mapping from input values to FRPs just like the wiring diagram does.

```

pgd> T
A conditional FRP with wiring:
  <0>: An FRP with value 0
  <1>: An FRP with value 1

```

However, when we *evaluate* a conditional FRP, we simulate the process of giving it an input and activating it; the input value is passed through and prepended to the value produced by the activated FRP.⁴⁷

```

pgd> T(0)
An FRP with value <0, 0>
pgd> T(1)
An FRP with value <1, 1>
pgd> T(2)
Value <2> not in the domain of this conditional FRP.

```

Given a value outside $\{0, 1\}$, T raises an error.⁴⁸ Let cons_c denote the “prepend $\langle c \rangle$ ” statistic, where $\text{cons}_c(v) = \langle c \rangle :: v$. Then, we can write

$$\begin{aligned}
 T(0) &= \text{cons}_0(N) \\
 T(1) &= \text{cons}_1(P).
 \end{aligned}$$

We form the *mixture* depicted in the previous example by connecting the output of D to the input of the conditional FRP T . We denote the resulting FRP by $D \triangleright T$, or $D \gg T$ in the playground.

⁴⁷Use `T.target(value)` or `T[value]` to get the *target* FRP without the input prepended.

⁴⁸Recall: We treat $T(\langle x \rangle)$ and $T(x)$ as equivalent. See Section F.7.

```
pgd> D >> T
An FRP with value <0, 0>
```

In this run of the playground, D has value 0 which is passed to T, giving T(0) as a result. (Of course, your values may differ for these FRPs.)

Having gotten a feel for conditional FRPs, their wiring and evaluation, we can now formalize the idea.

Definition 9. A **conditional FRP** is a function R that maps values in a finite set \mathcal{V} to FRPs. We call \mathcal{V} the *domain* of R and the corresponding FRPs the *targets*.

When given an input value $v \in \mathcal{V}$ and activated, R produces the concatenated value $v :: w$, where w is the value produced by the target in R wired to v .

We require that the target FRPs for a given dimension input all have the same dimension. If a conditional FRP accepts values of dimension m and target FRPs have dimension n , we say that the conditional FRP has

- **type** $m \rightarrow m + n$,
- **codimension** m , and
- **dimension** $m + n$.

Every FRP of dimension n is also a conditional FRP of type $0 \rightarrow n$.

The dimension is $m + n$ because the input value passes through and is prepended to the value produced by the activated FRP. That a regular FRP is just a conditional FRP of codimension 0 means that it needs no input to activate. An ordinary FRP still accepts a connection at its input port from an output port of another FRP for transforming with statistics and building mixtures, but this is a distinct phenomenon.

The words “conditional” and “condition” arise in several contexts throughout this material, which can be confusing. In this case, the word conditional is meant as in programming: “if we get a 0 then use N , else use P .” A conditional FRP (or Kind) is choosing an FRP (or Kind) contingently on some value. We will often express such contingency with the word “given”: given an input v , the result looks like this; given an input w , the result looks like that; and so on.

The conditional FRP T from Example 4.5 has type $1 \rightarrow 2$. Even though the constituent FRPs, N and P have dimension 1, the value produced by T *includes the input value* and so is two dimensional. This type tells us that the output of T can be connected to the input of another conditional FRP that accepts pairs of 0s and 1s.

Puzzle 29. In the playground, build a conditional FRP whose domain contains three values and that returns FRPs of at least two different Kinds.

For a conditional FRP R , we can take the Kind of each FRPs comprising it, which associates each input value to a Kind. The function r from v to $\text{kind}(R(v))$ returns a Kind for each input value v in the domain of R . We call this a *conditional kind*.

In Example 4.5, the conditional FRP T activates N given input $\langle 0 \rangle$ and P given input $\langle 1 \rangle$. The conditional Kind t associated with T is illustrated by the wiring diagram in Figure 31. This shows the Kind of the activated FRP given each valid input. Notice the direct correspondence between the wiring diagrams for t and T here and in Example 4.5.

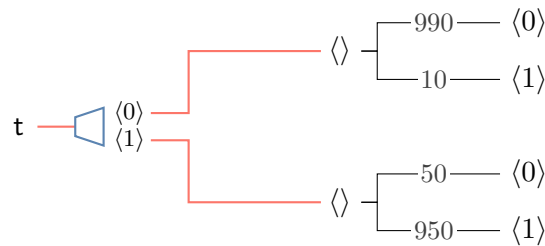


FIGURE 31. The conditional Kind t corresponding to the conditional FRP T in Example 4.5.

In the playground, we can also construct conditional Kinds directly in ways analogous to how we constructed conditional FRPs, using `conditional_kind` as a function or decorator. For instance, following up on our earlier example, we can use a dictionary mapping values to Kinds

```
pgd> t = conditional_kind({
...>     0: either(0, 1, 99),
...>     1: either(0, 1, 1/19)
...> })
```

or a named function

```
pgd> @conditional_kind(domain=[0, 1], target_dim=1)
...> def t(value):
...>     if value == 0:
...>         return either(0, 1, 99)
...>     return either(0, 1, 1/19)
```

or an anonymous function

```
pgd> t = conditional_kind(
...>     lambda value:
...>         either(0, 1, 99) if value == 0 else either(0, 1, 1/19),
...>     domain=[0, 1], target_dim=1
...> )
```

Try these and look at each `t`. The optional arguments `codim`, `dim`, `target_dim`, and `domain` are used as for `conditional_frp`. It is good practice to specify these (especially `codim` and either `dim` or `target_dim`) when building a conditional Kind from a function so that errors can be raised on invalid input or operations.

Note that just as for a conditional FRP, when you look at `t` in the playground, it shows you the mapping of values to Kinds, like the wiring diagram in Figure 31.

```
pgd> t
A conditional Kind with wiring:
      ,---- 0.99000 ----- 0
<0>:  <> -|
      `---- 0.010000 ---- 1

      ,---- 0.050000 ---- 0
<1>:  <> -|
      `---- 0.95000 ----- 1
```

This shows the Kind of the FRP attached to each input wire. But when you *evaluate* `t`, it gives the Kind of the *value produced by the conditional Kind when given that input* to which the input value has been prepended.⁴⁹

```
pgd> t(0)
      ,---- 0.99000 ----- <0, 0>
<> -|
      `---- 0.010000 ---- <0, 1>
pgd> t(1)
      ,---- 0.050000 ---- <1, 0>
<> -|
      `---- 0.95000 ----- <1, 1>
```

⁴⁹Use `t.target(value)` or `t[value]` to get the *target* Kind without the input.

The wiring diagram makes the construction of the system clearer, but the evaluated Kinds show what is actually used by the system. Observe that $t(0) = \text{kind}(T(0))$ and $t(1) = \text{kind}(T(1))$.

Definition 10. A **conditional Kind** is a function r that maps values in a finite set \mathcal{V} to Kinds. We call \mathcal{V} the *domain* of r and the corresponding Kinds the *targets*.

When given an input value $v \in \mathcal{V}$, r produces a Kind based on the target but with the input v *prepended* to every leaf node w as $v :: w$.

We require that the target Kinds for a given dimension input all have the same dimension. If a conditional Kind accepts values of dimension m and if the returned Kinds have dimension n , we say that the conditional Kind has

- **type** $m \rightarrow m + n$,
- **codimension** m , and
- **dimension** $m + n$.

Every Kind of dimension n is also a conditional Kind of type $0 \rightarrow n$.

Every conditional FRP R has an associated conditional Kind r with the same domain defined by

$$r(v) = \text{kind}(R(v)) \quad (4.8)$$

for every v in the domain of R . Thus, $r = \text{kind} \circ R$ is a composition of functions, “kind after R ,” where kind maps FRPs to their Kinds. This is good, but it is useful to extend the kind function to *also* accept conditional FRPs and return their corresponding conditional Kind. So, we can write $r = \text{kind}(R)$ to express the relationship, and in the playground, we can use $\text{kind}(R)$ to find the corresponding conditional Kind.

```
pgd> kind(T)
```

```
A conditional Kind with wiring:
```

```

      ,---- 0.99000  ----- 0
<0>:  <> -|
      `---- 0.010000 ----- 1
      ,---- 0.050000 ----- 0
<1>:  <> -|
      `---- 0.95000  ----- 1
```

```
pgd> conditional_kind({ 0: kind(N), 1: kind(P) })
```

Both of these are the same object as what we got for t earlier.

Puzzle 30. In the playground, build a conditional Kind to describe the following system: flip three coins and take the sum of h balanced, six-sided dice, where h is the number of heads flipped.

It will likely be easiest to define this with a named function decorated by `@conditional_kind`. If you want to specify a domain, you can pass one of the following as the `domain` argument:

```
((i,j,k) for i in [0,1] for j in [0,1] for k in [0,1])
including the parentheses or, after first entering, import itertools,
itertools.product([0,1], repeat=3)
```

4.3 General Mixtures

For many random systems/processes, the evolution of the system at later stages is contingent on what happens at earlier stages. To describe such systems and build FRPs that represent them, it is generally easier to take a modular approach: describe individual stages and the FRPs that represent them and connect those stages together to describe and represent the system as a whole. This is what mixtures are for.

Mixtures allow us to build FRPs in stages where the value produced at one stage determines the FRP used at the next. When we combine FRPs this way, the values from earlier stages pass through and are combined with the values produced at later stages, so the values of a mixture FRP produces records the outcomes at every stage. Independent mixtures are a special case⁵⁰ where the stages do not interact; there is no contingency. We get independent mixtures by wiring each mixer to targets with *the same kind*. General mixtures come from allowing the targets to be FRPs with different Kinds.

⁵⁰ An important special case.

Conditional FRPs (and their conditional Kinds) package up all the targets of a mixture into a single device. The input port only accepts inputs in a designated set, and given an input value, an activated conditional FRP will display a value, the input value combined with whatever the activated target produces. These can then be the main ingredients in mixtures.

We say that two conditional FRPs R and S or two conditional Kinds r and s are **compatible** if all possible values of R (r) belong to the domain of S (s). That is,

all possible values of the former are valid inputs to the latter. This implies that the dimension of R (r) equals the codimension of S (s).


We get a mixture of two compatible conditional FRPs by connecting the output of one to the input of the other. The result is another conditional FRP.

Definition 11. If R and S are compatible conditional FRPs of respective types $m \rightarrow n$ and $n \rightarrow p$, then their **mixture** is the conditional FRP $R \triangleright S$ of type $m \rightarrow p$. This is obtained by connecting the output of R to the input of S .



The domain of $R \triangleright S$ equals the domain of R , and given input v , its value is the value of $S(R(v))$.

In particular, if R is an ordinary FRP of dimension n (i.e., has type $0 \rightarrow n$) then the mixture $R \triangleright S$ is an ordinary FRP of dimension p (i.e., has type $0 \rightarrow p$).

We depict a conditional FRP as  in wiring diagrams, but usually dropping the selector switch when the codimension is 0, i.e., for an ordinary FRP.

Keep in mind that the value produced by a conditional FRP that is given an input value and activated in a mixture is the concatenation of the input value and the value produced by the activated target. This is the value that is passed on to the next stage of a mixture. So given input v , $R \triangleright S$ produces the value of $S(R(v))$. Let's trace this through. $R(v)$ obtains the value w of the FRP in R wired to v and produces the concatenated tuple $v :: w$. This is then passed as input to S , which obtains the value x of its FRP wired to $v :: w$ and produces the concatenated tuple $v :: w :: x$. This value includes the input and the value produced by each constituent FRP activated in the mixture.

If R is an ordinary FRP (i.e., has codimension $m = 0$), then $R \triangleright S$ is also an ordinary FRP. Its value concatenates the value of R and the contingent value of S given the value of R . In Example 4.5, for instance, we defined an FRP D that represents whether the patient has the disease and a conditional FRP T that represents the outcome of the test contingent on whether the patient has the disease. The mixture $D \triangleright T$ is an FRP that represents both the patient's disease status and test outcome. When we activated this FRP in the playground on page 153, this FRP

had value $\langle 0, 0 \rangle$, indicating that the patient does not have the disease and tested negative.

Example 4.6. Waiting at the airport, my flight has been delayed several times, and I worry that the delays will get worse. I can switch to a later flight to my destination, hoping that the later flight leaves before my original (delayed) flight. Or I can stick with my original flight hoping that whatever problem is causing the delay is resolved before the later flight is scheduled to leave.

Let A be the conditional FRP that represents the difference between my actual and scheduled arrival times at my destination, contingent on my choice of flight. This returns a different FRP depending on whether I choose the later or original flight.

Lacking any useful information, I choose the flight by a coin flip, an FRP F with Kind

$$\langle \rangle \text{ --- } \begin{cases} 1 \text{ --- } \langle 0 \rangle \\ 1 \text{ --- } \langle 1 \rangle \end{cases}$$

The mixture $F \triangleright A$ is an FRP of dimension 2, with first component indicating the flight I chose and the second component indicating the delay in my arrival time. The transformed FRP $\text{proj}_2(F \triangleright A)$ represents the delay in my arrival time with my chosen flight.

Example 4.7. One piece of pizza remains at the FRP Warehouse office party, and three friends each want it for themselves. Unwilling to divide it, they devise a scheme to select fairly among the three at random using only the large supply of FRPs with Kind

$$\langle \rangle \text{ --- } \begin{cases} 1 \text{ --- } \langle 0 \rangle \\ 1 \text{ --- } \langle 1 \rangle \end{cases}$$

that are piled in a corner of the room.

Let values 1, 2, and 3 label the three friends, and value 0 indicate that a decision has not yet been made. They start with an FRP C_0 that is a constant with value 0 and an FRP representing a balanced coin flip.

```
pgd> C_0 = frp(constant(0))
pgd> Flip = frp(either(0, 1))
```

They then construct a conditional FRP:

```
pgd> F_1 = conditional_frp({
  0: clone(Flip) * clone(Flip) ^ (2 * Proj[2] + Proj[1]),
  1: frp(constant(1)),
  2: frp(constant(2)),
  3: frp(constant(3))
})
```

If they have decided who gets the pizza, F_1 returns an FRP whose value is that choice. But if they have not decided, F_1 represents a transform of two coin flips giving value 0 for $\langle 0, 0 \rangle$, 1 for $\langle 0, 1 \rangle$, 2 for $\langle 1, 0 \rangle$, and 3 for 0 for $\langle 1, 1 \rangle$. This value represents the choice of pizza winner. Because all four possibilities for two balanced coin flips have the same weight, the three friends are fairly selected, *except* there is still the possibility (value 0) where the choice is not resolved. Will the pizza go to waste? They find

```
pgd> F_1
A conditional FRP with wiring:
<0>          An FRP with value 0
<1>          An FRP with value 1
<2>          An FRP with value 2
<3>          An FRP with value 3
pgd> C_0 >> F_1
An FRP with value <0, 0>
pgd> C_1 = Proj[2](C_0 >> F_1)
An FRP with value <0>
```

The mixture selects from F_1 the FRP corresponding to C_0 's value (which we know to be 0) and concatenates the value of C_0 with the value of that FRP. Had that FRP had value 1, 2, or 3, then the FRP C_1 would give the choice

among the three friends for who gets the pizza. But alas, it has value 0, which means that the choice is not yet resolved. So they do it again!

Because they want a conditional FRP “F_2” that looks just like F_1 but with different clones, it is easier if they define a *factory* for the conditional FRP as follows:

```
pgd> def F_n():
    return clone(F_1)
```

and using F_n() gives a fresh conditional FRP with the right structure. Now,

```
pgd> C_2 = Proj[2](C_1 >> F_n())
An FRP with value <0>
```

The drama continues.

```
pgd> C_3 = Proj[2](C_2 >> F_n())
An FRP with value <2>
```

So friend 2 eats the pizza. But was this really a fair process?

```
pgd> kind(C_0)
<> ----- 1 ----- 0
pgd> kind(C_1)
,---- 1/4 ----- 0
|---- 1/4 ----- 1
<> -|
|---- 1/4 ----- 2
`---- 1/4 ----- 3
pgd> kind(C_2)
,---- 1/16 ----- 0
|---- 5/16 ----- 1
<> -|
|---- 5/16 ----- 2
`---- 5/16 ----- 3
pgd> kind(C_3)
```

```

,---- 0.015625 ---- 0
|---- 0.32813 ----- 1
<> -|
|---- 0.32813 ----- 2
`---- 0.32813 ----- 3

```

The weights on 1, 2, and 3 are equal at every stage, and after repeating the process n times, the weight on 0 is 4^{-n} , which gets small quickly, so after at most a few iterations, they are likely to resolve the choice. Friends 1 and 3 may be glum, but they cannot feel mistreated.

Puzzle 31. You have two FRPs X and Y , and you want to construct their independent mixture $X \star Y$ using the mixture operator \triangleright .

Describe a conditional FRP U that makes $X \triangleright U = X \star Y$.

Puzzle 32. Characters in *Dungeons & Dragons* who specialize in combat (called “fighters”) and who roll an 18 for their Strength attribute score can make an additional roll of a balanced 100-sided die to determine their “exceptional strength” sub-score in the range $[1..100]$. All other characters have exceptional strength 0.

Modify the FRP factory `dnd_character` in Example 4.1 to include an exceptional strength sub-score in the value. The FRP returned by this factory will have dimension 7. The factory should be defined with an optional argument that indicates if the character is a “fighter,” `False` by default (e.g., `def dnd_character(fighter=False)`).

Notice that there are different Kinds of mixtures at play here because the exceptional strength sub-score depends on the character’s Strength attribute.

A roll for exceptional strength is an FRP whose Kind can be generated in the playground with `uniform(1, 2, ..., 100)`.

Puzzle 33. If S, T, U are scalar FRPs and $W = S \star T \star U$, what are the components of W ?

(The components of an FRP are described in Definition 7 on page 62.)

Puzzle 34. If X has components $\langle X_1, X_2, \dots, X_n \rangle$, is it always true that we can write $X = X_1 \star X_2 \star \dots \star X_n$? If so why? If not, can you find an example where this relationship does not hold?

The strong correspondence that we have seen several times so far between operations on FRPs and operations on Kinds holds also for mixtures. Just as we can build mixtures of conditional FRPs, we can build mixtures of conditional Kinds.

Definition 12. If r and s are compatible conditional Kinds of respective types $m \rightarrow n$ and $n \rightarrow p$, then their **mixture** is the conditional Kind $r \triangleright s$ of type $m \rightarrow p$.

We form the mixture as follows: For each valid input v to r and for each leaf node w in $r(v)$, attach the tree $s(w)$ to that leaf node. This gives a combined tree for every v .

In particular, if r is an ordinary Kind of dimension n (i.e., has type $0 \rightarrow n$) then the mixture $r \triangleright s$ is an ordinary Kind of dimension p (i.e., has type $0 \rightarrow p$).

The mixture operations on conditional FRPs and on conditional Kinds are compatible:

$$\text{kind}(R) \triangleright \text{kind}(S) = \text{kind}(R \triangleright S). \quad (4.9)$$

So, the mixture of Kinds equals the Kind of the mixture

In Example 4.5, let $d = \text{kind}(D)$, the Kind of the FRP D representing whether the patient has the disease, and let t be the conditional Kind defined earlier. Figure 32 shows the process for building the mixture Kind $d \triangleright t$. For each value v of d , we take the Kind $t(v)$ and attach it at the corresponding leaf node of d . On the left, the Figure shows the component Kinds; on the right, it shows mixture Kind, which we can convert to canonical form.

```
pgd> kind(D) >> kind(T)
,---- 0.98901 ----- <0, 0>
|---- 0.0099900 ----- <0, 1>
<> -|
|---- 0.000050000 ---- <1, 0>
`---- 0.00095000 ----- <1, 1>
pgd> FRP.sample( 1_000_000, D )
```

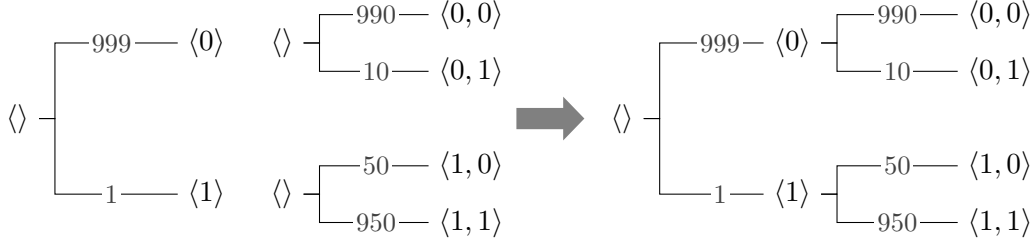


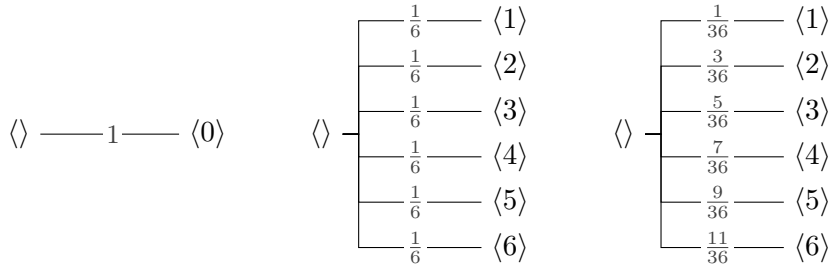
FIGURE 32. Constructing the Kind mixture $d \succ t$. For each value v of d , we take the Kind $t(v)$ and attach it at the corresponding leaf node of d , forming the Kind tree at right.

Summary of output values:

$\langle 0, 0 \rangle$	988786 (98.88%)
$\langle 0, 1 \rangle$	10176 (1.02%)
$\langle 1, 0 \rangle$	44 (0.00%)
$\langle 1, 1 \rangle$	994 (0.10%)

We are close here to answering the original question in the example, and will see a nice way to do it exactly in the next section. But for now, we can see an approximate answer here; 994/10176 is the proportion of positive tests among those with the disease.

As another example of taking mixtures of Kinds, suppose we flip two balanced coins (equal weight on heads and tails) and given h heads, take the maximum of h rolls of a balanced six-sided die. (If $h = 0$, we take the maximum to be 0.) The FRPs representing each of the coin flips has Kind $\langle \rangle \begin{array}{l} \frac{1}{2} \text{---} \langle 0 \rangle \\ \frac{1}{2} \text{---} \langle 1 \rangle \end{array}$ where 0 means tails and 1 means heads. The FRPs representing the three possible rolls have Kinds



The Kind describing the outcome of the system is a mixture: we take an independent mixture of the coin flip Kind with itself and then mix with a conditional Kind that

gives the Kind of the roll for each outcome of the coin flips. This is easier to see in the playground.

```
pgd> flip = either(0, 1)           # kind of a coin flip
pgd> roll0 = constant(0)           # kind of no rolls
pgd> roll1 = uniform(1, 2, ..., 6) # kind of one roll
pgd> roll2 = Max(uniform(1, 2, ..., 6) ** 2) # kind of max of two rolls
```

Notice that in `roll2` we use an independent mixture of the single-roll Kinds before transforming with the statistic. Look at these Kinds in the playground and compare to the pictures above. The contingent roll is described by a conditional Kind:

```
pgd> roll = conditional_kind({
...>      (0, 0): roll0,
...>      (0, 1): roll1,
...>      (1, 0): roll1,
...>      (1, 1): roll2
...> })
```

Now we can form the mixture, two independent flips and a contingent roll:

```
pgd> outcome = flip ** 2 >> roll
pgd> unfold(outcome)
```

Look at the canonical and unfolded trees in the playground; the latter reveals the steps of the mixture. First, a copy of `flip` is attached to each leaf node of `flip`, which is the independent mixture. Then, a copy of `roll0`, `roll1`, or `roll2` is attached to each leaf of the tree produced by the first stage, depending on how many values. Notice how the history of the process is recorded in the values at each node. The unfolded Kind is shown in Figure 33.

Keep in mind that the independent mixture `flip ** 2` forms the first two stages of the whole mixture. This is in fact equivalent to the general mixture `flip >> conditional_kind(flip, codim=1)` because for Kind `k`, `conditional_kind(k)` gives a conditional Kind with target `k` for any input. So we could write `outcome` with just the `>>` operator by

```
flip >> conditional_kind(flip, codim=1) >> roll
```

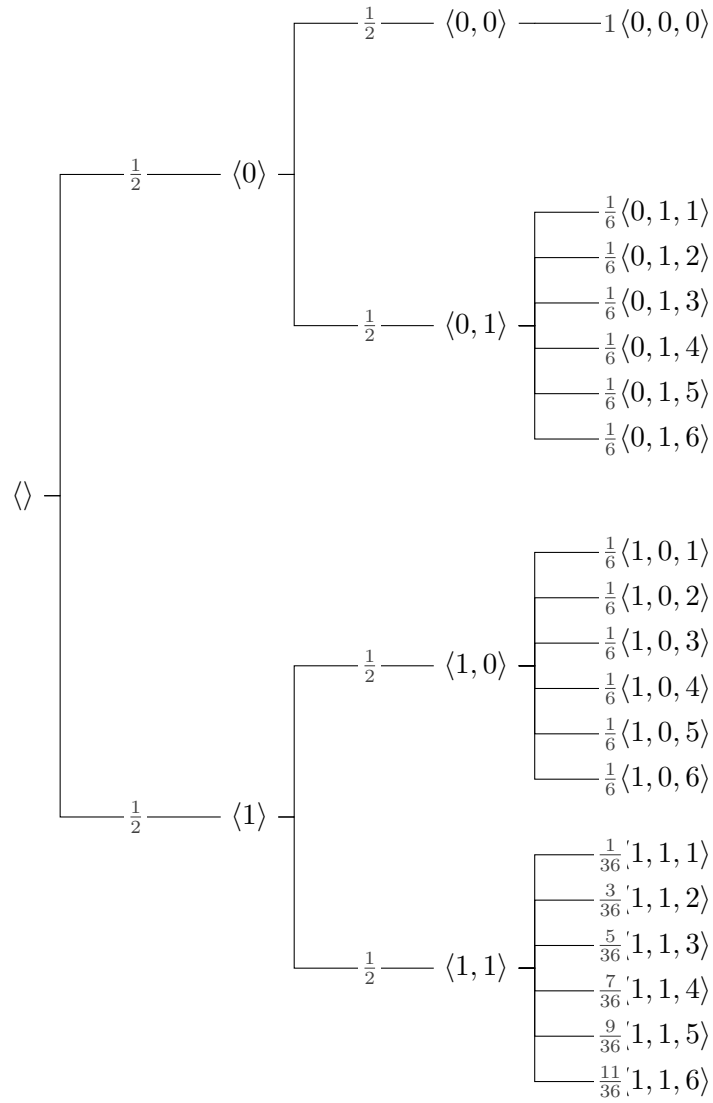



FIGURE 33. The unfolded mixture Kind representing two coin flips and a contingent dice roll.

Puzzle 35. Rewrite the conditional Kind `roll` in terms of a named function

```
@conditional_kind(codim=2)
def roll(flips):
    ...
```

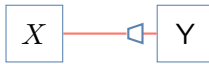
that returns a Kind. You can enforce a domain constraint by using

```
@conditional_kind(codim=2, domain=lambda v: all(x in {0,1} for x in v))
```

as the decorator above to test that the input tuple consists of only 0s and 1s.

Next, we consider several examples that illustrate these ideas. These examples nicely illustrates the contingent evolution that mixtures capture and how we can use mixtures to describe complicated systems.

Example 4.8 Random Point in a Circle We want to generate a point with integer coordinates at random from a circle of radius 5, and we would like all the points to be chosen with equal weight. We will build an FRP to do this in two stages: X will be an FRP that represents the x -coordinate, and Y will be a conditional FRP that chooses the y -coordinate contingent on the x -coordinate.



We will build this directly in the playground. Let's start with what seems like the obvious solution (but which fails): choosing x - and y -coordinates with equal weights subject to the constraint that the mixture value lies in the circle.

```
pgd> X = frp(uniform(-5, -4, ..., 5))
pgd> @conditional_frp(domain=irange(-5, 5), target_dim=1)
...> def Y(x):
...>     y_kind = uniform(y for y in irange(-5, 5)
...>                     if y * y <= 5 * 5 - x * x)
...>     return frp(y_kind)
pgd> kind(X >> Y)
```

You will notice that this Kind does *not* put equal weight on all the values. Why not? Consider the two values $\langle 0, 0 \rangle$ and $\langle 5, 0 \rangle$. Look at the unfolded Kind tree with `unfold(kind(X >> Y))` and reconstruct the weights in the canonical form. Does that suggest a better solution?

Let's define two small helper functions:

```
pgd> def y_points(x, r=5):
...>     return [y for y in irange(-r, r) if y * y <= r * r - x * x]
pgd> num_y_points = compose(len, y_points)    # len `after` y_points
```

Now, we can adjust our first approach slightly to solve our problem.

```
pgd> X = frp(weighted_by(-5, -4, ..., 5, weight_by=num_y_points))
pgd> @conditional_frp(domain=irange(-5, 5), target_dim=1)
...> def Y(x):
...>     return frp(uniform(y_points(x)))
pgd> XY_m = X >> Y
pgd> kind(XY_m)
```

Here, we give weight to x -coordinates in proportion to the number of points at that x -coordinate that lie inside the circle. The Kind (of size 81) has weights on all random points equal to $1/81$.

The mixture $X \triangleright Y$ is an FRP that represents a random choice of integer points inside a circle (of radius 5), where all points are chosen with equal weight.

We could construct this in another way:

```
pgd> points_inside_5 = [(x, y) for x in irange(-5, 5)
...>                     for y in irange(-5, 5)
...>                     if x * x + y * y <= 25]
pgd> XY_j = frp(uniform(points_inside_5))
pgd> Kind.equal( kind(XY_j), kind(X >> Y) )
True
```

The two constructions – building with a mixture (`XY_m`) and specifying all components jointly (`XY_j`) – give the same results, but each has practical advantages in some circumstances. Mixtures break a problem into smaller pieces that are

easier to describe, and they often scale well computationally. These advantages make mixtures our most common way to build more complex systems. However, modeling components jointly allows direct specification of the weights, which is often useful. For instance, we can easily give points weight that decreases with their distance from the origin with

```
pgd> point_wgts = [numeric_exp(-(x*x + y*y)) for x,y in points_inside_5]
pgd> XY_jd = frp(weighted_as(points_inside_5, weights=point_weights_5))
pgd> kind(XY_jd)
```

This can be done with mixtures but takes a bit more effort.

With either construction, we can predict the answers to questions about the random point. For instance: How far away is the point from the origin? (The statistic `Norm` computes the root sum of squares of the components.)

```
pgd> kind(Norm(XY_m))
,---- 0.012346 ---- 0
|---- 0.049383 ---- 1
|---- 0.049383 ---- 1.4142
|---- 0.049383 ---- 2
|---- 0.098765 ---- 2.2361
|---- 0.049383 ---- 2.8284
|---- 0.049383 ---- 3
<> -|
|---- 0.098765 ---- 3.1623
|---- 0.098765 ---- 3.6056
|---- 0.049383 ---- 4
|---- 0.098765 ---- 4.1231
|---- 0.049383 ---- 4.2426
|---- 0.098765 ---- 4.4721
`---- 0.14815 ----- 5
pgd> E(Norm(XY_m))
3.391780659838882
```

Or: Is X bigger than Y ?

```

pgd> compareXY = IfThenElse(Proj[1] > Proj[2], 1,
...>                        IfThenElse(Proj[1] < Proj[2], -1, 0))
pgd> kind(compareXY(XY_m))
      ,---- 0.45679 ----- -1
<> -+----- 0.086420 ----- 0
      `----- 0.45679 ----- 1

```

Observe that, as we might expect, X and Y are equally likely to be biggest.

You can load FRP factories for this example with

```
om frplib.examples.circle_points import circle_points
```

Then `circle_points()` return a clone of `XY_j`, and also accepts a different circle radius. (The helpers `y_points`, `num_y_points`, and `points_inside`, which takes an optional radius, are also available.) We will use `circle_points` in ensuing examples.

Example 4.9 Random Lines

We now want to generate a random *line* formed by two random points within the circle of radius 5 generated as in the previous example. Our only constraint is that the two points not be the same, so the line is well defined.

```

pgd> First_Point = circle_points()
pgd> point_kind = kind(First_Point)
pgd> @conditional_frp(domain=points_inside_5, target_dim=2)
...> def Second_Point(first_point):
...>     not_same_point = (__ != first_point) # a statistic
...>     return frp(point_kind | not_same_point)
pgd> Line = First_Point >> Second_Point
pgd> Line
An FRP with value <1, -4, -4, 3>

```

The conditional FRP `Second_Point` takes a point as input and generates a random point with equal weight inside the circle *excluding the input point*. To do this, it defines a Boolean statistic (aka. a condition) that tests whether a given point is equal to the input point. It then uses the `|` operator, which is read as

“given,” to impose a constraint: `point_kind | not_same_point` is a Kind like `point_kind` that excludes the input point. (We will discuss this in detail in the next section.) The mixture generates the first point, passes it to the conditional FRP to generate the second point, and the result is the concatenation of the two points: a line from $\langle 1, -4 \rangle$ to $\langle -4, 3 \rangle$.

What can we say about how long the generated line is?

```
pgd> line_length = Norm(Proj[1,2] - Proj[3,4])
pgd> kind(line_length(Line))
,---- 0.043210 ----- 1
|---- 0.040741 ----- 1.4142
|---- 0.037654 ----- 2
|      ...           ...
<> -|
|      ...           ...
|---- 0.0024691 ---- 9.4868
|---- 0.0012346 ---- 9.8995
`---- 0.0018519 ---- 10
pgd> E(line_length(Line))
4.667555258539501
```

where some output has been omitted. (Look at it in the playground.) Notice how we build the statistic `line_length`. We take the first two components as a vector tuple and the last two as a vector tuple, subtract them as vectors (componentwise), and then take the length of the result. The resulting Kind has properties we would expect: a minimum length of 1, for adjacent points, and a maximum length of 10 along a diameter of the circle. Roughly speaking, longer lines are less likely because there are fewer ways to obtain them, and our best prediction of the line length ≈ 4.67 is slightly less than half the maximum.

Example 4.10 Headphone Interface

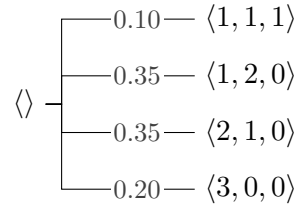
I like my wireless headphones well enough, but they have only a single control button: push to play, push to stop. If I want to advance or rewind a track or

episode, I need to double-click or triple-click that button. (If at the beginning of a track, a triple click moves to the beginning of the previous track; otherwise, it moves to the beginning of the current track.) Unfortunately, double or triple clicking a button in one's ear is highly unreliable because the earbud squishes around as the button is pressed.

I am listening to a track and want to repeat it from the beginning, so I attempt to triple click the control button. The clicks register *randomly* with the headphones as either a triple click, a double click and a click, a click and a double click, or three triple clicks. This leaves me at the beginning of the target track, as desired, or paused at the beginning of the following track, or paused where I was. If paused, I hit the button to continue. What is the chance that I am able to rewind the target track to the beginning within four attempts? (After four attempts, I give up and grudgingly open my phone to restart the track directly.)

Let's build a system to model this situation and answer my question. We will represent the possible outcomes of an attempted triple click with 3-tuples: $\langle 3, 0, 0 \rangle$ for a triple click, $\langle 2, 1, 0 \rangle$ for a double click then a single click, $\langle 1, 2, 0 \rangle$ for a single click then a double click, and $\langle 1, 1, 1 \rangle$ for three single clicks. The 0's here are just placeholders that keep the a constant dimension for the output.

Based on my experience, assume that the outcome of an attempted triple click has Kind K_1 given by



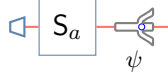
We will also use the constant Kind K_0

$$\langle \rangle \text{ --- } 1 \text{ --- } \langle 0, 0, 0 \rangle$$

for FRPs that always produce the value $\langle 0, 0, 0 \rangle$. Recall also that the empty FRP, **empty**, is reconfigured to display the output of another FRP that is connected to its input port, as described in Section 2.

Before solving this in the playground, it is worth sketching out the structure of our system. As the system evolves, we will keep track of two numbers $\langle t, a \rangle$, where t describes the current track and a counts the number of triple-click attempts so far. The track can have value in the set $\mathcal{T} = \{0, \frac{1}{2}, 1, 2, 3, 4\}$. The values 0-4 denote the beginning of successive tracks, with 0 meaning the target track I am listening to initially. The value $1/2$ means that the target track is in progress, but not at the very beginning. The attempts records how many times we push the button before achieving our goal or giving up; it has value in $[0..4]$. The tuple $\langle t, a \rangle$ comprises the *state* of the system, and each time we attempt a triple click, the state is randomly updated. We *start* in state $\langle \frac{1}{2}, 0 \rangle$, with the target track in progress.

For each $a \in [0..4]$, we define a conditional FRP S_a . For $a = 0$, this has type $0 \rightarrow 2$ and is an FRP that always returns the initial state $\langle \frac{1}{2}, 0 \rangle$. The FRP S_0 represents the initial state of the system. For $a \in [1..4]$, S_a has type $2 \rightarrow 5$. It accepts inputs $\langle 0, b \rangle$ for $0 \leq b < a$ and $\langle t, a - 1 \rangle$ for $t \in \mathcal{T}$ and $0 < t < a$, the former when having successfully rewound on the b th attempt and the latter all other possible states after $a - 1$ attempts. S_a connects those inputs to FRPs with Kind K_0 if $t = 0$ or K_1 if $t > 0$.



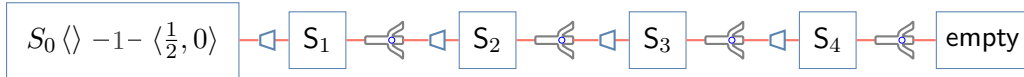
For $a > 0$, we then transform the output of S_a with the statistic ψ defined for valid $\langle t, a \rangle$ with $t > 0$ by

$$\begin{aligned}\psi(0, a, x, y, z) &= \langle 0, a \rangle \\ \psi(t, a, 3, 0, 0) &= \langle \lceil t \rceil - 1, a + 1 \rangle \\ \psi(t, a, 2, 1, 0) &= \langle \lfloor t \rfloor + 1, a + 1 \rangle \\ \psi(t, a, 1, 2, 0) &= \langle \lfloor t \rfloor + 1, a + 1 \rangle \\ \psi(t, a, 1, 1, 1) &= \langle t, a + 1 \rangle,\end{aligned}$$

where ceiling $\lceil t \rceil$ is the smallest integer $\geq t$ and floor $\lfloor t \rfloor$ is the greatest integer $\leq t$. The transformed conditional FRPs have type $2 \rightarrow 2$. If we reach $t = 0$, we

stop counting attempts.

Taken together, our system is wired as follows:



The output values $\langle t, a \rangle$ either have $t = 0$ if I am able to rewind successfully or $t > 0$ and $a = 4$ otherwise.

Now, let's build this in the playground. We start by defining with the Kinds K_0 and K_1 and the initial state FRP S_0 :

```
pgd> K_0 = constant(0, 0, 0)
pgd> K_1 = weighted_as((3, 0, 0), (2, 1, 0), (1, 2, 0), (1, 1, 1),
...>                  weights=[0.2, 0.35, 0.35, 0.1])
pgd> S_0 = frp(constant('1/2', 0))
```

Here, `constant` is the factory for the constant Kind with only the given value, and `weighted_as` creates a Kind with arbitrary values and specified weights. Note that we can give fractional quantities as strings; for some fractions (though not $1/2$), this gives a more accurate numerical representation internally.

We define the conditional FRPs S_a for $a \in [1..5]$ in two steps: the base mapping common to all four conditional FRPs and a factory function that enforces the requirements on the inputs.

```
pgd> def S_base(v):
...>     t, a = v
...>     if t == 0:
...>         return frp(K_0)
...>     return frp(K_1)

pgd> def S_(a):
...>     "Returns the conditional FRP S_a for a in 1..4."
...>     assert a in set([1, 2, 3, 4])
...>     domain_S = ( [(0, b) for b in range(a)] +
```

```

...> [(t, a - 1) for t in [0.5, 1, 2, 3, 4] if t < a] )
...>
...> return conditional_frp(S_base, domain=domain_S)
pgd> S_(1)
A conditional FRP as a function with domain={(0.5, 0), (0, 0)}
pgd> S_(2)
A conditional FRP as a function with domain={(0, 1), (0.5, 1), (1, 1), (0, 0)}

```

We can get away without the factory, but the domain constraints do help us avoid mistakes. And remember that either way, we need *distinct* conditional FRPs for each of the four stages.

Next, we define the statistic ψ :

```

pgd> @statistic(codim=5, dim=2)
...> def psi(v):
...>     match v:
...>         case (0, a, _, _, _):
...>             return (0, a)
...>         case (t, a, 3, 0, 0):
...>             return (numeric_ceil(t) - 1, a + 1)
...>         case (t, a, 2, 1, 0) | (t, a, 1, 2, 0):
...>             return (numeric_floor(t) - 1, a + 1)
...>         case (t, a, 1, 1, 1):
...>             return (t, a + 1)
...>     raise MismatchedDomain(f'Improper input {v} to psi')

```

For Python versions before 3.10, `match` is unsupported, so you can use explicit if-then-else statements.

Finally, we put this all together into an FRP that represents the entire system. We write this as a simple factory to return a fresh FRP:

```

pgd> def try_rewind():
...>     "Returns an FRP that represents trying to rewind the current track."
...>     S = S_0
...>     for a in irange(1, 4):
...>         S = S >> S_(a) ^ psi
...>     return S

```

```
pgd> try_rewind()
An FRP with value <3, 4>
pgd> try_rewind()
An FRP with value <0, 2>
```

The loop in `try_rewind` exactly expresses our wiring diagram above. We feed S_0 into S_1 and through ψ , and the value of the resulting FRP becomes the input for the next stage. On the first try here, we give up; on the second, we succeed on the second attempt. We can compute the Kind of this FRP to assess our chances:

```
pgd> kind(try_rewind())
,---- 0.2 ----- <0, 1>
|---- 0.16 ----- <0, 2>
|---- 0.03 ----- <0, 3>
|---- 0.0240 ----- <0, 4>
<> -+---- 0.0001 ----- <1/2, 4>
|---- 0.04200 ----- <1, 4>
|---- 0.16660 ----- <2, 4>
|---- 0.13720 ----- <3, 4>
`---- 0.24010 ----- <4, 4>
pgd> kind(try_rewind() ^ (Proj[1] == 0))
,---- 0.5860 ----- 0
<> -|
`---- 0.414 ----- 1
```

The statistic `Proj[1] == 0` tests whether we end up at the beginning of the target track, so the second Kind tells us we have roughly 41% chance of succeeding. Put another way, computing our prediction:

```
pgd> E(try_rewind() ^ (Proj[1] == 0))
0.414
```

We can also ask where we will end up after this try.

```

pgd> kind(Proj[1](try_rewind()))
,---- 0.414 ----- 0
|---- 0.0001 ---- 1/2
|---- 0.042 ----- 1
<> -|
|---- 0.1666 ---- 2
|---- 0.1372 ---- 3
`---- 0.2401 ---- 4

```

So, we are likely to end up several tracks ahead.

Note that we need not write a factory like `try_rewind`, though it is convenient. We could express our wiring diagram in a single expression, with a few extra parentheses to avoid ambiguity:

```

pgd> (((S_0 >> S_(1) ^ psi) >> S_(2) ^ psi) >> S_(3) ^ psi) >> S_(4) ^ psi
An FRP with value <0, 4>

```

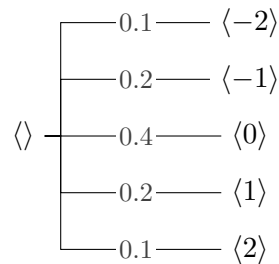
The parentheses are needed because the `^` operator has lower precedence than `>>`; without them, `psi` would group improperly with the following `S_(a)`.

Example 4.11 The Drunken Sailor

A drunken sailor stands on a dock facing the gangway taking him to his ship. If he moves five steps forward, he makes it onto the ramp and, however unsteadily, to safety on the ship. If he moves five steps backward, he falls off the dock into the ocean and ends up sleeping on the beach as his ship leaves without him. Assume that he moves forward (1) or backward (-1) at random and independently at each step, where his direction has Kind

$$\langle \rangle \begin{cases} \xrightarrow{\frac{3}{5}} \langle -1 \rangle \\ \xrightarrow{\frac{2}{5}} \langle 1 \rangle \end{cases}$$

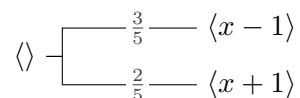
Assume the dock is 9 steps long and that we label his positions on the dock by integers in $[-4..4]$. When the sailor stumbles down to the dock, he initially moves onto a starting position randomly, with Kind



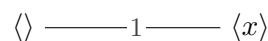
What is the probability that he makes it to the ship within 10 lurching steps? To reasonable approximation, what is the probability that he eventually makes it to the ship without falling into the ocean? We can visualize this system with the wiring diagram



where we can extend with more conditional FRPs as desired. Here, D is an FRP representing the sailor's starting position on the dock, with the Kind above. The conditional FRPs S_i all have the same structure. They map a position x in $[-4..4]$ to an FRP with Kind



and a position x in $\{-5, 5\}$ to a constant FRP with Kind



Let's compute the answers to our question in the playground

```
pgd> move_kind = either(-1, 1, '3/2')
pgd> D = frp(weighted_as(-2, -1, ..., 2, weights=[0.1, 0.2, 0.4, 0.2, 0.1]))
```

We define a single conditional FRP S and use `clone(S)` to make the copies S_1, S_2, \dots, S_{10} . We first define a Boolean function that tests for valid inputs;

giving this function as the `domain` argument to `conditional_frp` raises an error with any input for which this function returns False.

```
pgd> def is_valid_path(v):
...>     valid_positions = set(irange(-5, 5))
...>     return all(x in valid_positions for x in v)

pgd> @conditional_frp(domain=is_valid_path, target_dim=1)
...> def S(path):
...>     x = path[-1]                # most recent position
...>     if x == -5 or x == 5:        # boat or ocean...
...>         return frp(constant(x))  # ...stay there
...>     return frp(move_kind ^ (__ + x)) # move left or right
```

Following the wiring diagram, we start with D and successively mix with a copy of S in a loop. The final value of W is an FRP representing the sailor's path over 10 steps.

```
pgd> W = D
pgd> for _ in range(10):
...>     W = W >> clone(S)

pgd> W
An FRP with value <0, 1, 0, -1, -2, -3, -4, -5, -5, -5, -5>
pgd> clone(W)
An FRP with value <0, -1, -2, -3, -4, -5, -5, -5, -5, -5>
pgd> clone(W)
An FRP with value <2, 1, 0, 1, 2, 3, 4, 3, 4, 5, 5>
pgd> clone(W)
An FRP with value <1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1>
pgd> ten_steps = kind(Proj[-1](W))
```

We can see on running the system anew several times that sometimes the sailor falls in the ocean, sometimes reaches the boat, and sometimes neither within 10 steps. We can view the Kind of W easily, but since it has size 3902, you should look at it on your terminal rather.

More interestingly, we can predict the answer our various questions about the sailor's trajectory. Does the sailor reach the boat or the ocean in 10 steps?

```
pgd> ten_steps ~ Or(Proj[-1] == 5, Proj[-1] == -5)
      ,---- 0.65354 ---- 0
<> -|
      `---- 0.34646 ---- 1
pgd> which_end_point = Cases({-5: -5, 5: 5}, default=0)(Proj[-1])
pgd> which_end_point(ten_steps)
      ,---- 0.29466 ----- -5
<> -+---- 0.65354 ----- 0
      `---- 0.051806 ---- 5
```

Here, we use the *statistic combinator* `Or` to compute the logical-or of two statistics. (For internal Python reasons, we cannot use Python's `or` keyword in this context.) We see that the sailor has only about a 0.35 probability of reaching an end point in 10 steps, and is about 6 times as likely for that endpoint to be ocean rather than boat if he does.

There is nothing magical here about 10 steps. We can extend this further, for example to sixteen steps, but because we are tracking the sailor's entire path, the number of possibilities – and thus the size of the Kind – gets large quickly:

```
pgd> sixteen_steps = kind(W >> S >> S >> S >> S >> S >> S)
pgd> size(sixteen_steps)
185502
pgd> sixteen_steps ~ Or(Proj[-1] == 5, Proj[-1] == -5))
      ,---- 0.43221 ---- 0
<> -|
      `---- 0.56779 ---- 1
```

This leaves a relatively high probability that the situation will be unresolved. We want to simulate enough steps that to good approximation, we can assume he will reach one endpoint or another.

Fortunately, for many questions, we do not need the sailor's entire path, only selected information about it. We can follow the approach used in [Example 4.10](#),

where we transform the output of the conditional FRP by a statistic that keeps the dimension fixed and tracks the information we care about. (We adapt this approach to the next example as well.)

We will keep track of the sailor's current position x and the number of *steps* n he has made. But if the sailor reaches ocean (-5) or boat (5), the value stays as is. We tweak our conditional FRPs to take a tuple $\langle x, n \rangle$ as input; their values will look like $\langle x, n, x', n' \rangle$ where x' is the sailor's next position and n' is an updated count.

```
pgd> @conditional_frp(target_dim=2)
...> def S2(v):
...>     x, n = v
...>     if x == -5 or x == 5:
...>         return frp(constant(x, n))
...>     return frp(move_kind ^ Fork(__ + x, n + 1))
```

We extend D 's value with an initial number of steps (0) as the sailor's initial state. We get the sailor's next state ($\langle x', n' \rangle$) from the FRP by applying a projection to extract the last two components of the value. We package this in a function that has the maximum number of steps as a parameter:

```
pgd> def sailors_walk(up_to_step):
...>     W2 = D ^ Fork(Id, 0)
...>     for _ in range(up_to_step):
...>         W2 = W2 >> clone(S2) ^ Proj[3,4]
...>     return W2

pgd> kind(sailors_walk(16)) ^ Or(Proj[1] == 5, Proj[1] == -5)
,---- 0.43221 ---- 0
<> -|
`---- 0.56779 ---- 1
```

which is just what we got above but about a thousand times faster.

This lets us compute the outcome for longer walks in which the sailor is essentially certain to reach one of the endpoints.


```
pgd> kind(sailors_walk(200)) ^ Or(Proj[1] == 5, Proj[1] == -5)
,---- 9.9261E-7 ---- 0
<> -|
`---- 1.00000 ----- 1
```

So to good approximation, the sailor's will reach an endpoint within the first 200 steps.

```
pgd> many_steps = kind(sailors_walk(200))
pgd> many_steps ^ IfThenElse(Abs(Proj[1]) < 5, 0, Proj[1])
,---- 0.86986 ----- -5
<> -+---- 9.9261E-7 ---- 0
`---- 0.13014 ----- 5
```

So to good approximation, the sailor is about 7.5 more likely to end up in the ocean than on the boat.

We can predict how many steps it will take the sailor to reach either end:

```
pgd> E(Proj[2](many_steps))
18.49313503683753
```

Here, `Proj[2](many_steps)` is the Kind of the second component of the FRPs value, the number of steps taken. This answer is a bit too high because it includes the very small chance remaining that the situation is not resolved. In the next section, we will see how to impose a conditional constraint on the output with the `|` operator, which is read “given”.

```
pgd> E(Proj[2](many_steps | (Abs(Proj[1]) >= 5)))
18.49295487075902
```

The condition after the given `|` is the constraint we require. *Given* that the sailor reaches the boat or the ocean, how many steps does it take? The prediction is only very slightly different, as expected. Later, we will also see how to use these Kinds to solve this problem exactly.

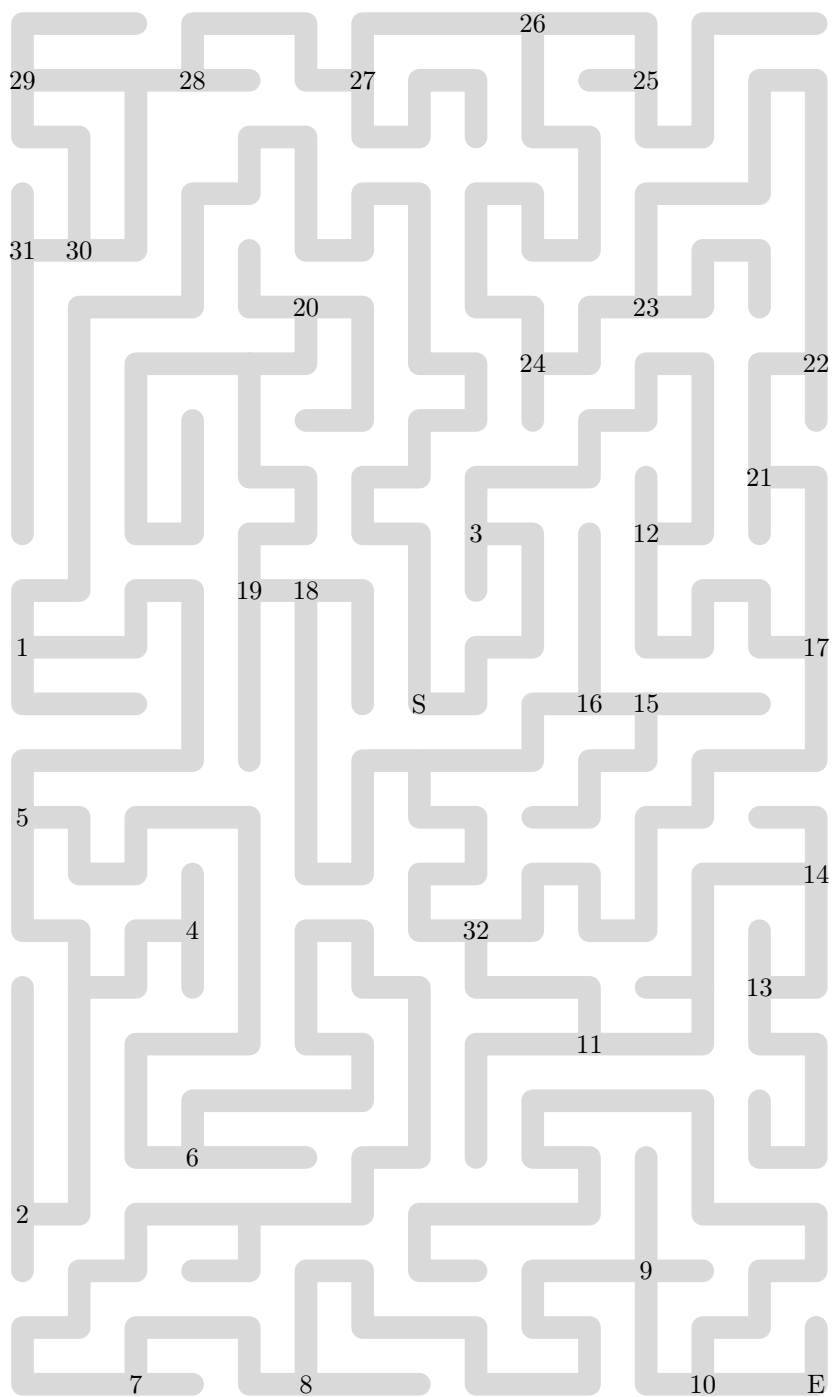


FIGURE 34. Another labyrinth that has ensnared poor Theseus. His starting point ($S=0$) and exit (E) are marked, and each juncture is assigned a number. The FRPs will generate a number corresponding to the juncture that Theseus wanders into.

Example 4.12 Back to the Labyrinth

Theseus has awoken from a night of revelry to find himself trapped in a labyrinth ... again (Figure 34). With both the excess of honey mead and the lack of Ariadne's help, he is not at his best, and he wanders about at random from his starting position, looking for the exit.

Because our FRPs generate lists of numbers, our first step is to assign a number to each relevant outcome. Here, the key information is which junctures in the labyrinth Theseus visits. So we assign a unique number to each juncture, as shown in the Figure.

When Theseus stands at juncture 17, for example, he has three choices (move to junctures 4, 18, 19), and in his stupor he chooses from among them randomly with equal weight (`uniform(4, 18, 19)` in the playground). The same applies at every juncture, beginning with the starting point 0.

Open the playground and follow along. We start by creating the Kind of the FRP for Theseus's starting position. He begins at juncture 0 with certainty, so this is a constant.

```
pgd> start = constant(0)
```

Then, we import some data about the labyrinth so that you do not have to type it all in. The variable `labyrinth` contains a dictionary mapping each juncture to the junctures Theseus can reach from it. Take a look at its value and see how it corresponds to the Figure.

```
pgd> from frplib.examples.labyrinth import *  
pgd> labyrinth
```

Notice that juncture 33 is the exit, and even in his diminished capacity, Theseus will take the exit when he gets there. So, `labyrinth[33] = [33]` to reflect that he exits when he reaches juncture 33.

We want to use `labyrinth` to generate mixtures, and this is easy to do. We start by creating the Kinds for Theseus's moves at each juncture. For each "item" in the labyrinth – a juncture and its list of neighbors – we associate with that juncture a Kind that gives equal weight to every neighbor (via the `uniform`

factory). The call to the `conditional_kind` factory gives `steps` some useful properties and makes it print out nicely.

```
pgd> steps = conditional_kind({
...>     juncture: uniform(neighbors)
...>     for juncture, neighbors in labyrinth.items()
...> })
pgd> moves = from_latest(steps)
```

In Python, this is called a *dictionary comprehension*.

Specifically, `steps` maps each juncture number to the Kind for a move out of that juncture. Look at its value in the playground. The function `from_latest` is defined in the `frplib.examples.labyrinth` module converts the conditional Kind `steps` that takes as input a single juncture to a conditional Kind that accepts as input a path of any length. As the name suggests, the latter uses only the latest juncture in path, so `moves` takes a *tuple* describing Theseus's path so far and uses `steps` to make a move based on the last juncture in the path.

Consider the FRPs describing Theseus's path after one, two, and three moves. Take a moment and think about how to get their Kinds from `start` and `moves`.

Puzzle 36. We could generate a table of FRPs at each juncture like

```
pgd> steps_at = conditional_frp({
...>     juncture: frp(kind)
...>     for juncture, kind in steps.items()
...> })
```

Why isn't this sufficient to simulate Theseus's trip through the maze? Hint: Once you push the button on an FRP, can the value change?

Given a path to any juncture, `moves` returns the Kind for a move *from* that juncture, so that will go on the right-hand side of `>>`. This yields

```
pgd> start                                # starting position
pgd> start >> moves                        # after one move
pgd> start >> moves >> moves               # after two moves
pgd> start >> moves >> moves >> moves      # after three moves
```

The `>>` operator is left-associative, so without parentheses, an expression like the last line groups from the left automatically:

```
((start >> moves) >> moves) >> moves
```

Look at these Kinds. At each leaf, we see – for that particular random outcome – Theseus’s entire path through the maze so far. The results show the Kind trees in canonical form; you can always use `unfold` to see the full tree, e.g., `unfold(start >> moves)`, though these can get big.

Puzzle 37. How would you find the Kind of the FRP describing Theseus’s path through $n \geq 0$ moves? Write or sketch a function `n_moves` that takes an initial Kind and n and a conditional Kind like `moves` and returns the corresponding Kind when you call `n_moves(start, n, moves)`.

For large numbers of moves, the Kind trees get large because the FRP’s value can reflect many possible paths. Of course, in Theseus’s besotted state, he is not thinking too clearly, and he is making moves that ignore his previous path. So, we will use a trick to make things more manageable: we will look at the Kind of his *most recent move* only. That is, we will create an FRP that answers the question *at what juncture is Theseus after 100 moves?*

The function `after_move_n` has been loaded into your playground to help with this. Let’s simulate Theseus’s 100th move, passing `steps` (not `moves`).

```
pgd> move100_kind = after_move_n(100, start, steps)
pgd> frp(move100_kind).value
pgd> FRP.sample(1000, frp(move100_kind))
```

The first line gives the Kind of an FRP that records just Theseus’s 100th move. The second line gives an FRP with that Kind and pushes the button. The third line generates 1000 FRPs with that Kind and summarizes the result.

Puzzle 38. Use a sample of FRPs to estimate how likely Theseus is to have exited the labyrinth after 10 moves, 50 moves, 100 moves, and 1000 moves.

A key feature of a mixture is that its value includes the value of both the mixer and the target. In the previous examples, mixtures can construct an FRP (and its associated Kind) that represents the random process’s entire evolution up to some

point. In Example 4.12, for instance, the function `n_moves` forms the mixture Kind

```
start >> moves >> moves >> ... >> moves
```

that describes Theseus’s entire path from the start through a specified number of moves. In Example 4.11,

```
D >> S >> clone(S) >> ... >> clone(S)
```

is the mixture FRP that describes the sailor’s history of lurching steps. Mixtures join parts into a whole, describing the joint evolution of the parts as a random system.

But in many practical cases, we do not need to the entire history in our analysis, and as will be seen in the next section, often we only need to pay attention to the *most recent* state of the process to predict the next state. Indeed, this is what we did in each of the last three examples. The statistic ψ in Example 4.10 updated the track and number of attempts, dropping other information. The `Proj [3,4]` in `sailors_walk` of Example 4.11 saves only the sailor’s current position and number of steps. And in `theseus_latest`, we use a projection to extract Theseus’s last position (which is used in `after_move_n`) because Theseus’s next position only depends on where he is, not on how he got there.

Suppose we wanted to know the Kind of Theseus’s *second* position in the labyrinth without knowing his first position. We could do this in two steps (though one expression in the playground) with

```
pgd> second_pos_kind = (start >> moves)[2]
```

The first step forms the *mixture* of his first position and the conditional Kind of his second position given his first position. The second step is to transform with a projection that extracts only the second position, dropping the first.

If we wanted to know the Kind of Theseus’s *third* position in the labyrinth without knowing his second position, we use the same operation.

```
pgd> third_pos_kind = (second_pos_kind >> moves)[2]
```

Again, we form the mixture of the second position and the third position *given* the second position and then use a projection to extract the third position.

Mixtures build combined FRPs/Kinds, and projection statistics extract *marginals*, the FRPs/Kinds of selected components. Thus:

```

pgd> combined = start >> moves >> moves
pgd> Kind.equal( combined[1], start )
True
pgd> Kind.equal( combined[2], second_pos_kind )
True
pgd> Kind.equal( combined[3], third_pos_kind )
True

```

As we have seen, this is a common pattern, a special case of the *data-question* pattern of Figure 13: build a combined system as a mixture of several pieces and then extract marginals that relate to our questions. Call this the *mixture-marginal* pattern.

The Kinds representing Theseus's positions⁵¹ are instances of the mixture-marginal pattern with two pieces: his current position and a description of his next position that depends explicitly on his current position. This special case of the mixture-marginal pattern can be viewed in a different light that gives us the powerful *conditioning pattern*.

To understanding the conditioning pattern, we start with an FRP X (or its Kind) and a conditional FRP that we will call, for the purpose of this discussion, $\mathbf{YgivenX}$ (or its conditional Kind). The unusual name for $\mathbf{YgivenX}$ is intended to evoke the idea that it describes a second random quantity Y in a way that *depends on the value of X* . If $\dim(X) = m$ and $\mathbf{YgivenX}$ has type $m \rightarrow m + n$, then

$$Y = \text{proj}_{(m+1)..(m+n)} (X \triangleright \mathbf{YgivenX}) \quad (4.10)$$

$$\text{kind}(Y) = \text{proj}_{(m+1)..(m+n)} (\text{kind}(X) \triangleright \text{kind}(\mathbf{YgivenX})) . \quad (4.11)$$

This is just the mixture-marginal pattern, but here is where we shift perspectives, in two steps.

First, use the right-hand sides of these equations to define the **conditioning operator** // (written `//` in the playground) by

$$\mathbf{YgivenX} \text{//} X := \text{proj}_{(m+1)..(m+n)} (X \triangleright \mathbf{YgivenX}) \quad (4.12)$$

$$\text{kind}(\mathbf{YgivenX}) \text{//} \text{kind}(X) := \text{proj}_{(m+1)..(m+n)} (\text{kind}(X) \triangleright \text{kind}(\mathbf{YgivenX})) , \quad (4.13)$$

⁵¹The Kind of Theseus is something else entirely, where we replace branches successively with identical looking ones over time and ask if it is still the same Kind.

where $:=$ emphasizes that this is a definition. Equations (4.10) and (4.11) become

$$Y = Y_{\text{given}X} // X \text{ kind}(Y) = \text{kind}(Y_{\text{given}X}) // \text{kind}(X).$$

The conditional FRP or Kind goes on the left side of the operator, and the FRP or Kind of the quantity it depends on goes on the right side. This is the opposite of how we write the mixture because we are emphasizing the extracted quantity Y . In the Theseus example, `second_pos_kind` and `third_pos_kind` can be written, respectively, as

```
move // start           # same as second_pos_kind
move // second_pos_kind # same as third_pos_kind
```

We specify the update mechanism (moves) first and the state to update (e.g., `start` or `second_pos_kind`) second.

Second, if we focus on equation (4.11) above and consider how the mixture operation works, we can view this combination of mixture and projection as a *weighted-averaging* operation: `kind(Y)` is a weighted average of the Kinds produced by `kind(YgivenX)` using the weights in `kind(X)`. This is illustrated in Figure 35.

In the top panel of the Figure, we *condition on*⁵² the middle Kind, producing the Kind on the right-hand side. This is just mixture followed by a projection. Look at these in the playground.

⁵²We are using “condition on” as a verb here to mean applying the conditioning operation.

```
pgd> x = either(0, 1, weight_ratio=2)
pgd> y_given_x = conditional_kind({
...>   0: weighted_as(0, 2, 4, 9, weights=[3, 1, 1, 2]),
...>   1: weighted_as(1, 2, 4,   weights=[1, 3, 2]),
...> })
```

The conditional Kind on the left of the top panel is `y_given_x`, and the Kind in the middle of the top panel is `x`. We can compute the right-hand side in two equivalent ways:

```
pgd> (x >> y_given_x)[2]
pgd> y_given_x // x
```

The output is omitted here, but you should try it. In particular, compare this Kind `x >> y_given_x` and make sure you see how you get the former from the latter.

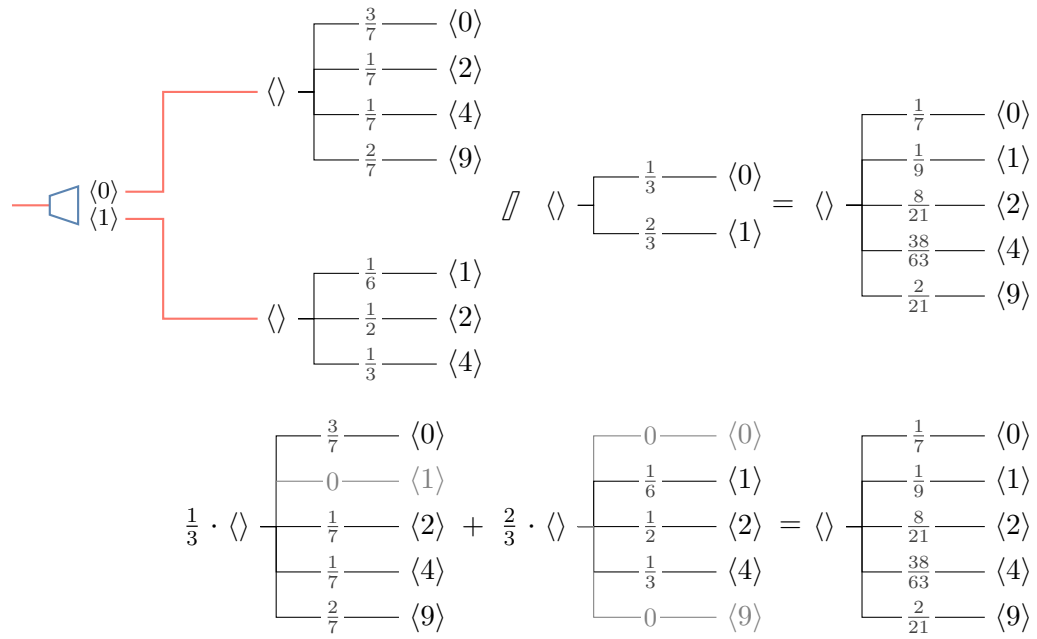


FIGURE 35. The conditioning operation as a weighted-average over a conditional Kind by a Kind. The top panel combines a conditional Kind and a Kind with the conditioning operator $//$, producing the Kind at the right. The bottom shows how that Kind can be viewed as an average of Kinds.

The bottom panel of the Figure gives us a different view of what this operation is doing. We first take each of the Kinds produced by `y_given_x` and add some fake, zero-weight branches to each so that both have the same set of values. We then average the weights of these Kinds, branch by branch, using the *corresponding weights from \mathbf{x}* . For instance, $\frac{1}{3} \cdot \frac{3}{7} + \frac{2}{3} \cdot 0 = \frac{1}{7}$ and $\frac{1}{3} \cdot \frac{1}{7} + \frac{2}{3} \cdot \frac{1}{3} = \frac{38}{63}$. This averaging of Kinds gives us the same result.

The operation of conditioning is just an instance of the mixture-marginal pattern. We mix, then project. But *conceptually*, we can view the conditioning operation as a way of finding the Kind of a random quantity by averaging over Kinds for the quantity that are specified conditionally on another random quantity. Each term in the average is weighted according to the Kind of that second quantity.

In Theseus’s case, `moves` specifies the Kind of his *next* position conditionally on his *current* position. If you know the Kind for his current position, you can find the Kind of his next position: `next = moves // current`. The `//` (and `///`) operator knows the dimension of the object on its right side and automatically tailors the projection to the dimension of the object on the left, which makes it convenient in the playground. For both Kinds or FRPs, the conditioning pattern looks like

$$y = y_given_x // x$$

The term on the left side is a conditional specification of y in terms of the quantity described by x . Later, we will also dub this the “method of hypotheticals” because we can use this to find the Kind of an FRP without actually observing the other quantity.

Definition 13. Suppose that X and Y are FRPs of dimension m and n , respectively, that are related by a mixture via

$$Y = \text{proj}_{(m+1)..(m+n)}(X \triangleright Y_{\text{given}X})$$

for some conditional FRP $Y_{\text{given}X}$ of type $m \rightarrow (m + n)$.

Then, equations (4.12) and (4.13) define the **conditioning** operator `//`, called the “conditioning” operator, which is written `//` in `frplib`.

The conditioning operator satisfies

$$Y = Y_{\text{given}X} // X \quad (4.14)$$

$$\text{kind}(Y) = \text{kind}(Y_{\text{given}X}) // \text{kind}(X). \quad (4.15)$$

We say here that we have computed Y (or its Kind) by “conditioning on” X (or its Kind). Equation (4.15) in particular says that we can compute the Kind of Y by combining a conditional Kind for Y ’s given X with the Kind of X .

Example 4.13. We flip a balanced coin. If it comes up tails, you roll a balanced six-sided die and take its value. If it comes up heads, you roll two balanced six-sided dice and take their maximum. If Y is an FRP whose payoff represents the value you take from this system, find $\text{kind}(Y)$ by conditioning?

Let X be the FRP representing the coin flip. We have a description of how Y depends on X and of X , so, we can compute $\text{kind}(Y)$ by conditioning on X

```
pgd> x = either(0, 1)    # kind(X)
pgd> y_given_x = conditional_kind({
...>    0: uniform(1, 2, ..., 6), 1: Max(uniform(1, 2, ..., 6) ** 2)
...> })
```

A conditional Kind with wiring:

,---- 1/6 ---- 1	,---- 1/36 ----- 1
---- 1/6 ---- 2	---- 3/36 ----- 2
---- 1/6 ---- 3	---- 5/36 ----- 3
<0>: <> -	<1>: <> -
---- 1/6 ---- 4	---- 7/36 ----- 4
---- 1/6 ---- 5	---- 9/36 ----- 5
`---- 1/6 ---- 6	`---- 11/36 ----- 6

```
pgd> y = y_given_x // x    # kind(Y) by conditioning on x
,---- 0.097222 ---- 1
|---- 0.12500 ----- 2
|---- 0.15278 ----- 3
<> -|
|---- 0.18056 ----- 4
```

```
|---- 0.20833 ----- 5
`---- 0.23611 ----- 6
```

Note for instance that the weight on 6 is $\frac{1}{2} \cdot \frac{1}{6} + \frac{1}{2} \cdot \frac{11}{36}$.

Answers to Selected Puzzles.

Puzzle 31. Independent mixtures are a special case: for any independent mixture, we can build it using the general mixture operation with proper choice of conditional FRP or Kind. For conditional Kinds s , we get an independent mixture with a *constant function* $s = \text{const}_k$ that always returns a Kind k . In this case, for a Kind r , $r \triangleright s = r \star k$. For conditional FRPs S , we get an independent mixture if S always returns an FRP with the same Kind. In particular, if $S = \text{const}_T$ for an FRP T , then for an FRP R , we have $R \triangleright S = R \star T$. So we choose $U = \text{const}_Y$.

Puzzle 32. We make a mixture from the strength attribute that introduces the exceptional strength, which is 0 except for a fighter with 18 strength.

```
@conditional_frp(domain=irange(3, 18), target_dim=1)
def extraStrength(strength):
    "Conditional kind for a fighter's exceptional strength."
    if strength == 18:
        return frp(uniform(1, 2, ..., 100))
    return frp(constant(0))

def dnd_character(fighter=False):
    "Returns an FRP representing a D&D character's attribute scores."
    # Strength with exceptional strength
    if fighter:
        S = dnd_attribute() >> clone(extraStrength)
    else:
        S = dnd_attribute() * frp(constant(0))
    I = dnd_attribute()    # Intelligence
    W = dnd_attribute()    # Wisdom
```

The constant function $\text{const}_c(x) = c$ is defined and discussed in Section F.2.

```

Co = dnd_attribute()    # Constitution
D = dnd_attribute()    # Dexterity
Ch = dnd_attribute()    # Charisma

return S * I * W * Co * D * Ch

```

We use `clone` so we get fresh FRPs each call for the exceptional strength.

Puzzle 35. We follow the template given:

```

@conditional_kind(domain=lambda v: all(x in {0, 1} for x in v))
def roll(flips):
    heads = sum(flips)
    if heads == 0:
        return constant(0)                # roll0
    return Max(uniform(1, 2, ..., 6) ** heads) # roll1 or roll2

```

Try entering `roll(0, 0)`, `roll(0, 1)`, and `roll(1, 1)` to check this.

Puzzle 36. As Theseus is wandering around the labyrinth, it is possible – even likely – that he will revisit the same juncture more than once. Each FRP has a single fixed value but Theseus makes a separate decision each time he visits. So more than one FRP per juncture may be needed.

Puzzle 37. We need to keep updating by mixing with `moves` n times, as follows:

```

def n_moves(start, n, moves):
    current = start
    for _ in range(n):
        current = current >> moves
    return current

```

Keep in mind though that the number of paths grows exponentially with n , so the tree gets *very big* rather quickly. We will see a way around that later.

Puzzle 38. For each $n = 10, 50, 100, 1000$, we do something like this

```

exit = 33
iter = 10000
nth = FRP.sample(iter, after_move_n(n, start, steps))
len([juncture for juncture in nth if juncture == exit])/iter

```

This computes the proportion of samples in which Theseus has reached the exit by move n . This works because once he reaches the exit, the FRP will always return that value.

Checkpoints

After reading this section you should be able to:

- Explain how to construct independent mixture of FRPs and of Kinds and what such mixtures mean.
- Describe what distinguishes an independent mixture from a more general mixture.
- Describe *conditional FRPs* and *conditional Kinds*, how to wire them together to form a mixture, and how to create them in the playground.
- Define the type, codimension, and dimension of a (conditional) FRP or Kind.
- Explain why an ordinary FRP or Kind can be consider a special case of a conditional FRP or Kind.
- Explain how to construct a general mixture between conditional FRPs or between conditional Kinds. In particular, given the tree from a Kind and the trees from the Kinds returned by a conditional Kind, show how to find the mixture Kind.
- Find the dimension, size, and values of the Kind $k \triangleright m$ from the properties of k and m .
- Recover X from $X \triangleright M$ or k from $k \triangleright m$ by applying an appropriate statistic.
- Relate the Kind of a mixture FRP X of an FRP and conditional FRP M to the mixture of the Kind $\text{kind}(X)$ and conditional Kind $\text{kind} \circ M$.

5 Constraining with Conditionals

Key Take Aways

A **conditional** applies a constraint from partial information about the value of an FRP. The result is a new FRP or Kind that captures the remaining uncertainty in the system and returns a value *consistent with the partial information*.

Conditionals frequently useful, including when we

- make observations *during* the evolution of random system and update our predictions accordingly;
- reason hypothetically about what our predictions *would be* if we had some particular knowledge;
- draw inferences from observed data to understand the structure of a random system; and
- decompose a random system into simpler but dependent pieces.

Conditional constraints are most useful when computing Kinds and expectations.

A **condition** is a Boolean-valued statistic. We use \top for true and \perp for false, or 1 and 0, as convenient. (We also use \top and \perp as a shorthand for the constant conditions const_{\top} and const_{\perp} .)

An **event** is an FRP that has only values 0 and 1. When an event has the value 1, we say that the event *occurred*, and when it has the value 0, we say that the event did not occur.

A **conditional constraint** is an event on the right side of the $|$ bar that implies that the object or expression on the left side of the $|$ should be interpreted as if **the event has in fact occurred**. When clear from context, it is sufficient to specify only a condition on the right side of the $|$ bar..

Applying a conditional constraint to a Kind means **erasing all branches in the tree whose values are inconsistent with the conditional constraint**. If K is a Kind and ζ is a compatible condition, then the Kind $K | \zeta$, read “ K given ζ ” is the tree obtained by eliminating all paths in K from root to leaf for which the leaf value v satisfies $\zeta(v) = \perp$.

If K is in canonical form, then $K | \zeta$ is obtained in canonical form by (i) erasing the branches of K whose values v satisfy $\zeta(v) = \perp$, and (ii) renormalizing the weights on the remaining branches to sum to 1.

If X is an FRP and ζ is a compatible condition, then $X \mid \zeta$ is the FRP obtained from X by forbidding any value v for which $\zeta(v) = \perp$. Note that if ζ is based on actual information observed about X 's value, then X and $X \mid \zeta$ will have the same value.

We have

$$\text{kind}(X \mid \zeta) = \text{kind}(X) \mid \zeta$$

and

$$\begin{array}{ll} K \mid \top = K & X \mid \top = X \\ K \mid \perp = \langle \rangle & X \mid \perp = \text{empty} . \end{array}$$

Bayes's Rule uses conditional constraints and projection statistics to infer earlier-stage components of a mixture from observations of later-stage components.

You run into Alice and Bob at the FRP Warehouse, and Alice is agitated. It seems that she had planned to place an order for a sizeable batch of b FRPs $A_{[1]}, A_{[2]}, \dots, A_{[b]}$ with the Kind shown in Figure 36. The deal stipulates that Alice will receive payoff from $P_{[i]} = \text{proj}_3(A_{[i]})$ for $i \in [1..b]$, paying \$ -0.825 per unit (i.e., the market is paying her). Anticipating her likely order – she is a regular customer – the Warehouse staff inadvertently presses the buttons on $A_{[1]}, \dots, A_{[b]}$ early, before she had committed to the deal. And Bob, who is wandering about the facility, happens to catch a glimpse of the displays. He tells Alice what he saw,⁵³ and Alice proceeds with the order at the original price – happily. When the Warehouse administrator hears what happened, she tries to rescind the order, and then ... let's just say that lawyers get involved. Why Alice was happy with the order and frustrated with the administrator's response?

⁵³Bob has a good recall, or a fast camera!

Some specifics: from his vantage point, Bob could see only the *second* number in the tuple displayed by each FRP $A_{[i]}$. He quickly tabulated his observations: out of 10,000 FRPs, he saw 4850 0's and 5150 1's. This is the information he gave Alice. This made Alice happy because she was convinced that the negotiated price for her order had become a bargain.

To understand Alice's reaction, we need to see how Bob's observations – partial information about the FRPs in the batch – changes Alice's predictions of her payoff.

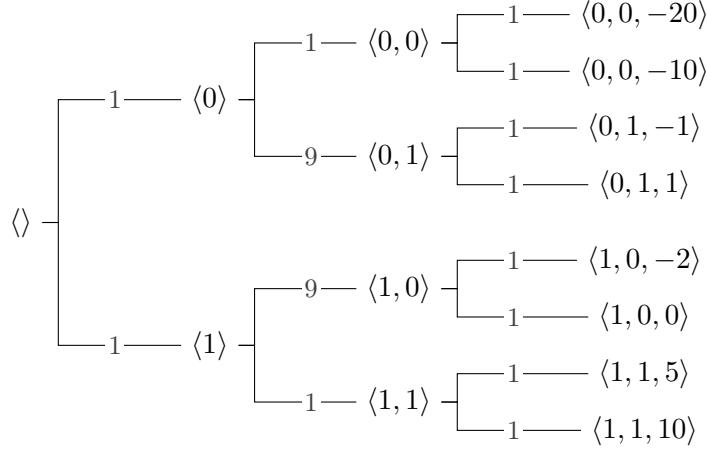
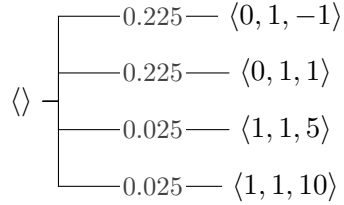


FIGURE 36. The Kind for the FRPs in Alice's order, *before* she learned Bob's information.

Once Alice obtains the knowledge of one component of an FRP's value, the FRP that determines her payoff has *effectively* been changed. Focus on a single FRP A with the same Kind (Figure 36) as $A_{[1]}, \dots, A_{[b]}$, and assume that you have *observed* (i.e., it's a fact) that $\text{proj}_2(A) = 1$. Although you do not observe the first or third components' values, you have nonetheless obtained information about them implicitly. We can see this by looking at all the paths in the Kind tree that *are consistent with the information you have*. The other paths are no longer relevant as they produce values inconsistent with *what we know to be true*. So to get the effective Kind we want to eliminate those paths from consideration. See Figure 37.

The situation is even clearer if we first reduce to canonical form, as shown in Figure 38. Both figures highlight the paths that are consistent with the available information. We simply *drop the inconsistent branches* of the tree to obtain the Kind of the effective FRP Alice has given the information about the second component.



A quick rescaling (useful but not required) returns us to canonical form, giving the Kind in the top panel of Figure 39. **This is the Kind of the FRP that Alice**

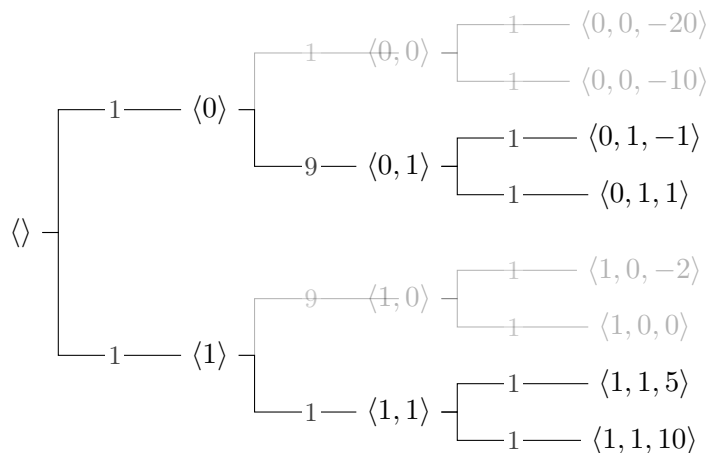


FIGURE 37. The Kind from the previous Figure, highlighting the paths in the tree that are consistent (black) and inconsistent (gray) with the observation that the second component of the value equals 1.

effectively gets *conditional* on the information that the second component is 1. In the playground, suppose `kindA` holds the original Kind, then we can compute our best prediction, or risk-neutral price, for the original FRP and for the effective FRP she is buying:

```
pgd> E(Proj[3]( kindA ))
-0.825
pgd> E(Proj[3]( kindA | (Proj[2] == 1) ))
3/4
```

The `|` operator here, read as “given,” introduces a *conditional constraint*, indicating that we should do our remaining calculations treating as a fact that the second component of the value equals 1. This increases the risk-neutral price by \$1.575.

A similar argument helps us evaluate the situation when Bob observe that the second component equals 0. The resulting Kind, in canonical form, is shown in the bottom panel of Figure 39. (You can obtain it by following the same procedure with different subtrees; just use the gray parts of the previous Figure.) Here

```
pgd> E(Proj[3]( kindA | (Proj[2] == 0) ))
-12/5
```

which is substantially below the original risk-neutral price.

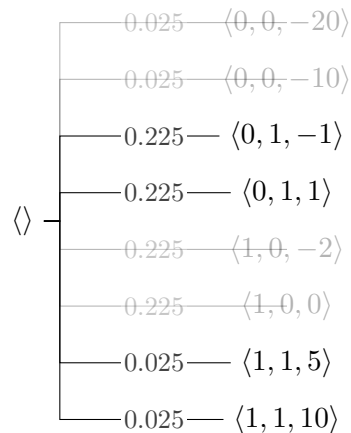


FIGURE 38. The canonical form of the Kind in the previous Figure, highlighting the values that are consistent (black) and inconsistent (gray) with the observation that the second component of the value equals 1.

Notice that all the values in Figure 39’s Kinds have the second component fixed at what it was observed to be. Only the other components of the value vary over the original possibilities, though some values are wholly eliminated.

Alice is being paid \$0.825 per FRP by the market in the original deal, but with Bob’s information, Alice she knows she will receive $10000 \cdot 0.825 + 4850 \cdot \frac{-12}{5} + 5150 \cdot \frac{3}{4} = 8250.00 - 7777.50 = 472.50$. She gets paid the agreed price per unit plus makes a profit in the values, yielding almost \$500. By trying to cancel the order on a technicality, the Market is trying to prevent a non-trivial payout to the client. Shady, Market. Very shady.

You can load `kindA` into the playground using

```
pgd> from frplib.examples.alice_bob import kindA
```

Use this with the following puzzle.

Puzzle 39. In the playground, build a Kind `kA` that is equal to `kindA`, and use either `inspection` or `Kind.equal(kA, kindA)` to check that you succeeded.

We can simulate Alice’s payoff from the FRPs she ordered with the FRP

```
P = Sum(frp(kindA[3]) ** 10_000)
```

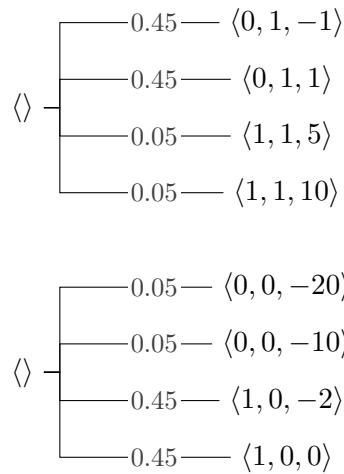


FIGURE 39. The *effective* Kind of Alice's FRPs conditional on the information that the second component of the value equals 1 (top) and 0 (bottom).

Explain briefly what this means.

In evaluating the Market manager's motives for trying to cancel the deal, we might wonder: "Is \$472.50 an unusually large payout for the original deal?"

Use `FRP.sample` with `P` and the statistic `(__ + 8250 >= 472.50)` to address the question. (One or two hundred samples should be sufficient to get an idea.)

When Bob told Alice the second component of her FRPs, that information allowed her to update her predictions about those FRPs' values. It is common in practice to obtain partial information about uncertain quantities as we observe a random system evolve, and it is often useful to determine how our predictions (and actions) would change *if* we had particular information. The purpose of **conditional constraints** is to account for partial information, observed or hypothetical, about the value of FRPs. If we have an FRP and observe something about its value, our predictions may change, and we want to adjust our calculations, decisions, and actions accordingly. In this section, we formalize what this means, how we represent and denote the conditional constraints, and how we use them to update our analysis. We will introduce conditional constraints in an expression with `|`, the conditional operator. This `|` bar is read "given" or "conditional on". On the left side of the `|` is the object that we are analyzing, on the right side is a specification of the partial information.

As mentioned earlier, the word "conditional" and its cognates get used a great

deal in probability theory, and at times, it might feel hard to keep track of them. We have already seen conditional FRPs, conditional Kinds, and conditions, and there will be more. The underlying theme is that an object is “conditional” if its value is contingent on the value of another object being known. So, a conditional FRP is an FRP that is contingent on the value produced by the FRP that is connected to its input, and a conditional constraint makes our analysis contingent on the truth of the specified partial information.

The first ingredient in a conditional constraint is a Boolean statistic that represents what the *condition* we take to be true in the constraint.

Definition 14. A **condition** is a statistic that returns a Boolean value.

A Boolean value is either \top , read “true” or “top”, or \perp , read “false” or “bottom”, though we often use 0 and 1 as synonyms for false and true, when convenient.

We also abuse notation a little bit by letting \top and \perp denote *constant* conditions. In this case, \top denotes the constant condition that returns true for *any* value and \perp denotes the constant condition that returns false for any value.

A condition describes how to determine whether the constraint is true for any input. In our computations, we often identify \top with the number 1 and \perp with the number 0, but we distinguish them notationally when we need to emphasize that Boolean and number are conceptually distinct types.

When we transform an FRP with a compatible condition, we get Boolean FRP that, as in the playground, we usually give the value 0 for false and 1 for true. FRPs that can have only the values 0 and 1 play a special role and have a name.

Definition 15. An **event** is an FRP that has only values 0 and 1.

When an event has the value 1, we say that the event *occurred*, and when it has the value 0, we say that the event did not occur.

We generally interpret the values of an event as false (0) and true (1), but we can operate on those values as numbers when convenient. Recall that we make no distinction between scalars and 1-tuples.

If we define For every event V , there is another *complementary event* that has the value false (0) when V has the value true (1) and the value true (1) when V has the

value false (0). We can define this as $\text{not}(V)$ with the statistic $\text{not}(v) = 1 - v$, but we usually write the statistic “inline”⁵⁴ as $1 - V$, or as $!V$ when we want to emphasize its Boolean-ness.

⁵⁴See discussion of inlined statistics on page 47.

We frequently specify events via **Boolean expressions** in terms of one or more FRPs. To understand this, we take in two steps, first we consider how to convert a Boolean expression into a condition, and then we apply that condition to an FRP. Recall that a Boolean expression is a mathematical statement in terms of one or more quantities that reduces to either true or false, or several such statements combined with logical-and (\wedge) or logical-or (\vee). For example, the expression $3x^2 - 4 > 0$ in terms of a numeric variable x will be true for some values of x and false for others. As discussed in detail in Section F.4, we can convert a Boolean expression into a function – called an indicator function or just *indicator* for short. An indicator is any function returns either 0 or 1, and the indicator for a Boolean expression returns 1 when the Boolean expression is true and 0 when it is false. The indicator for the expression $3x^2 - 4 > 0$ is a function, which we may write anonymously⁵⁵ as $\{3\blacksquare^2 - 4 > 0\}$, that returns 1 when given an x for which $3x^2 - 4 > 0$ is true and returns 0 when given an x for which $3x^2 - 4 > 0$ is false. *We write $\{3x^2 - 4 > 0\}$ for the value (0 or 1) returned when this indicator is evaluated at x .* Notationally, we surround the Boolean expression with braces, called **Iverson braces**, to indicate that we are extracting a Boolean value from the expression for the specified quantities.

⁵⁵See Section F.3 for more on anonymous functions. This is just a way to define a function without giving a name; we show where to put the argument in the returned expression.

A condition is just a special case of an indicator that accepts tuples of numbers as input. Each Boolean expression has an associated condition, so any Boolean expression given in terms of FRPs, like $\text{Sum}(A) = 1$ or $2 \text{proj}_1(A) + 7 > 10$, can be written as the transform of the FRPs by the associated condition, like $\zeta(A)$ or $\xi(A)$, where $\zeta(v)$ is true (1) when $\text{Sum}(v) = 1$ and $\xi(v)$ is true (1) when $2v_1 + 7 > 10$. In keeping with our notation for indicators, we can write the transformed FRPs inline as, e.g., $\{\text{Sum}(A) = 1\}$ or $\{2 \text{proj}_1(A) + 7 > 10\}$. These are just Boolean/ $\{0, 1\}$ -valued FRPs – that is, they are **events**. Notationally, we surround the Boolean expression with Iverson braces to indicate that we are transforming the FRPs by the associated condition. This is a special case of *inlining* a statistic as discussed on page 47. Note that if there is more than one FRP X, Y, Z, \dots in the Boolean expression, then we are applying the statistic to the combined FRP $X :: Y :: Z :: \dots$ that concatenates the values of its constituents.

In the Alice and Bob example, the information Bob learned about each of Alice’s

FRPs A was that $\text{proj}_2(A) = 1$, meaning that we have observed the second component of A to be 1. We thus observed that the event $\{\text{proj}_2(A) = 1\}$ occurred. Keep in mind that **events are FRPs**. They may or may not occur. And we can analyze events – e.g., compute their Kinds or expectation – just like any other FRP.

We specify a *conditional constraint* by giving an event on the right side of the given $|$ bar. This tells us to consider the object on the left side of the $|$ *assuming as a fact that the event on the right side has occurred*. If the input FRP to the event is understood, it is sufficient in practice to just use the condition to specify the constraint. This is common for instance in applying conditional constraints to a generic Kind and is how we specify conditional constraints in the playground.

Definition 16. A **conditional constraint** is an event on the right side of the $|$ bar that implies that the object or expression on the left side of the $|$ should be interpreted as if **the event has in fact occurred**.

When the input FRP is implicit or clear from context, it is sufficient to only put a condition to the right of the $|$, with the implied event being the transform of the input FRP by the given condition.

When an event specified in terms of a Boolean expression is given to the right of the $|$ bar, one can choose to omit the Iversion braces, as the expression’s interpretation as an event is clear.

In the basic case of conditional constraints, the given event is derived directly from the FRP or Kind being considered. If X is an FRP and ζ is a compatible condition, then $X | \zeta(X)$ is the FRP obtained from X by requiring the event $\zeta(X)$ to occur, i.e., that $\zeta(X)$ equals true (1). We write this as $X | \zeta$, read “ X *given* ζ ,” and call it an **FRP given a condition**. The understanding is that ζ transforms the FRP on the left of the $|$ to get the event that describes the conditional constraint. Think of $X | \zeta$ as a copy of X rewired to produce only values consistent with $\zeta(X)$ being true. If ζ is always true, then $X | \zeta$ equals X . The only case where this rewiring is not possible is when ζ is always false, then $X | \zeta$ equals the **empty** FRP, representing a logical contradiction.

In practice, we most often work directly with Kinds when computing with conditional constraints, and the impact of the conditional constraint on a Kind is more concrete. If K is a Kind and ζ is a compatible condition, $K | \zeta$, read “ K *given* ζ ,” and

called a **Kind given a condition**, is the tree obtained by eliminating all paths in K from root to leaf for which the leaf value v satisfies $\zeta(v) = \perp$. That is, we *eliminate* all branches in the canonical form of K that are *inconsistent with the condition being true*. We then usually renormalize the resulting Kind to canonical form, making the weights on the remaining branches sum to 1. If X is an FRP compatible with ζ , then

$$\text{kind}(X) \mid \zeta = \text{kind}(X \mid \zeta). \quad (5.1)$$

So we can apply a conditional constraint and compute the Kind in either order.

Definition 17. The Kind $K \mid \zeta$ is obtained in canonical form by (i) converting K to canonical form, (ii) erasing the branches of K whose values v satisfy $\zeta(v) = \perp$, and (iii) renormalizing the weights on the remaining branches to sum to 1.

For the condition \top that is always true, $K \mid \top = K$. For the condition \perp that is always false, $K \mid \perp = \langle \rangle$, the empty Kind, indicating a logical contradiction.

Example 5.1. An FRP representing three *independent* flips of a balanced coin (with 0 for tails and 1 for heads) has Kind $K = K_{\text{flip}} \star K_{\text{flip}} \star K_{\text{flip}}$, where K_{flip} is the Kind of a single flip. We will compute the Kind $K \mid \zeta$ for a variety of conditions, in the playground.

Puzzle 40. Define conditions that test the following assertions:

1. The first flip is a heads.
2. There is exactly one heads among the three flips.
3. There is at least one tails among the three flips.
4. The first and second flips have the same result.

As an example, the condition that tests if at most one of the first two flips is a heads can be defined as a named condition ψ by $\psi(v) = \{\text{proj}_1(v) + \text{proj}_2(v) \leq 1\}$ or as an anonymous condition by $\{\text{proj}_1(\blacksquare) + \text{proj}_2(\blacksquare) \leq 1\}$.

We start by defining the Kinds

```
pgd> flip = uniform(0, 1)
pgd> three_flips = flip ** 3
```



```

pgd> three_flips
,---- 1/8 ---- <0, 0, 0>
|---- 1/8 ---- <0, 0, 1>
|---- 1/8 ---- <0, 1, 0>
|---- 1/8 ---- <0, 1, 1>
<> -|
|---- 1/8 ---- <1, 0, 0>
|---- 1/8 ---- <1, 0, 1>
|---- 1/8 ---- <1, 1, 0>
`---- 1/8 ---- <1, 1, 1>

```

Next, we define several conditions to study. We show equivalent definitions of these conditions as named functions and as anonymous functions, where \blacksquare is a hole to be filled by the single argument.

When writing a hole by hand, use any consistent mark, like dash (–) or underscore (_).

Named Condition	Anonymous Condition
$\zeta(v) = \{\text{proj}_1(v) = 1\}$	$\{\text{proj}_1(\blacksquare) = 1\}$
$\xi(v) = \{\text{Sum}(v) = 1\}$	$\{\text{Sum}(\blacksquare) = 1\}$
$\varphi(v) = \{\text{Min}(v) = 0\}$	$\{\text{Min}(\blacksquare) = 0\}$
$\gamma(v) = \{\text{proj}_1(v) = \text{proj}_2(v)\}$	$\{\text{proj}_1(\blacksquare) = \text{proj}_2(\blacksquare)\}$

Using an anonymous condition, we can write $K \mid \zeta$ as $K \mid \text{proj}_1(\blacksquare) = 1$, for instance. Notice the similarity between the anonymous functions and the form of statistics/conditions in the playground, e.g., ζ written as $\{\text{proj}_1(\blacksquare) = 1\}$ is analogous to $(\text{Proj}[1] == 1)$.

```

pgd> three_flips | (Proj[1] == 1)
,---- 1/4 ---- <1, 0, 0>
|---- 1/4 ---- <1, 0, 1>
<> -|
|---- 1/4 ---- <1, 1, 0>
`---- 1/4 ---- <1, 1, 1>

```

In this case $K \mid \zeta$, we start with the Kind of K and eliminate all the branches v for which $v_1 \neq 1$. This leaves us the bottom subtree with the four branches whose first component is 1, all with weight $1/8$. Renormalizing the weights to sum to 1 gives the Kind shown, which is in canonical form. Notice that `(three_flips | (Proj[1] == 1)) ^ Proj[2,3]` is equal to `flip * flip`. (Try it!) Observing the first component of an independent mixture does not change our knowledge of the other components!

```
pgd> three_flips | (Sum == 1)
      ,---- 1/3 ---- <0, 0, 1>
<> -+---- 1/3 ---- <0, 1, 0>
      `---- 1/3 ---- <1, 0, 0>
```

In this case $K \mid \xi$, we again start with the Kind K and eliminate all the branches with other than one heads in three flips. This gives three branches, with the single heads in each component, all with equal weight, whose canonical form is as shown. Notice that

```
pgd> Kind.equal(kind(frp(K) | (Sum == 1)), K | (Sum == 1))
True
```

confirming equation (5.1).

```
pgd> three_flips | (Min == 0)
      ,---- 1/7 ---- <0, 0, 0>
      |---- 1/7 ---- <0, 0, 1>
      |---- 1/7 ---- <0, 1, 0>
<> -+---- 1/7 ---- <0, 1, 1>
      |---- 1/7 ---- <1, 0, 0>
      |---- 1/7 ---- <1, 0, 1>
      `---- 1/7 ---- <1, 1, 0>
```

Here, the condition only eliminates the branch $\langle 1, 1, 1 \rangle$ with no tails, leaving seven equally weighted branches.

```
pgd> three_flips | (Proj[1] == Proj[2])
,---- 1/4 ---- <0, 0, 0>
|---- 1/4 ---- <0, 0, 1>
<> -|
|---- 1/4 ---- <1, 1, 0>
`---- 1/4 ---- <1, 1, 1>
```

And here, we eliminate the four branches in K where the first two flips disagree, leaving four equally weighted branches remaining.

Puzzle 41. What do you expect to see when you enter

```
three_flips | (__ == (1, 0, 1))
```

in the playground? Explain why this makes sense.

In practice, we will often want to specify more general conditional constraints of the form $\text{kind}(X \mid V)$, where V is an event that may not be a direct transformation of X . Because the Kind given the constraint depends jointly on both X and V , to compute $\text{kind}(X \mid V)$, we need to start with an FRP that determines⁵⁶ the values of both X and V . Specifically, we require that D be an FRP such that $X = \psi_1(D)$ and $V = \psi_2(D)$ for some statistics ψ_1, ψ_2 . We define

$$\text{kind}(X \mid V) = \psi_1(\text{kind}(D) \mid \psi_2). \quad (5.2)$$

That is, we compute the $\text{kind}(D)$ given the conditional constraint that the event $V = \psi_2(D)$ occurs and transform that Kind by ψ_1 to focus on X , yielding the Kind of X given the event.

Let's see some examples in detail to make this concrete.

Example 5.2 What's in the Box?

We have three boxes, labeled 1, 2, and 3, each of which contains a number of colored balls. Box 1 contains 1 blue ball and 3 red balls; Box 2 contains 2 blue balls and 4 red balls; Box 3 contains 3 blue balls and 5 red balls.

I randomly choose a box and then randomly select a ball from that box,

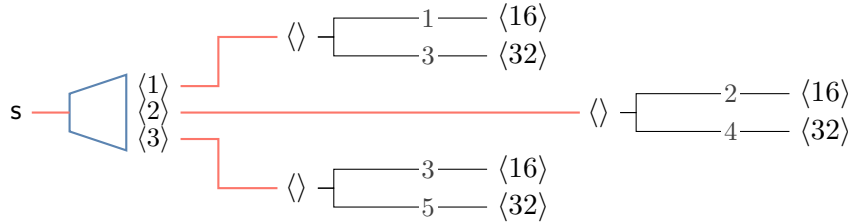
⁵⁶This is the role of the Data FRP in Figure 13, although we may use an FRP derived from that as well.

without showing you which box I chose. I then show you the color of the selected ball. Assume that all boxes are equally likely to be chosen and that all balls within the chosen box are equally likely to be selected. What have you learned about the chosen box from the color of the selected ball?

Let B be the FRP representing the chosen box, with values 1, 2, and 3. By our assumptions, B has Kind

$$\langle \rangle \begin{cases} \frac{1}{3} \text{---} \langle 1 \rangle \\ \frac{1}{3} \text{---} \langle 2 \rangle \\ \frac{1}{3} \text{---} \langle 3 \rangle \end{cases}$$

Let S be the conditional FRP of the selected ball given the chosen box, assigning arbitrary values 16 for a blue ball and 32 for a red ball. Based on the number of balls in each box and the assumption that each ball in the chosen box is equally likely to be selected, S has conditional Kind \mathbf{s} given by



The outcome of this random process is represented by the *mixture* FRP $B \triangleright S$, whose value specifies the chosen box and selected ball color. The FRP C , representing the color of the selected ball, is the second component of this mixture: $C = \text{proj}_2(B \triangleright S)$.

Observing a blue ball is represented by the event $\{C = 16\}$ occurring, or equivalently by the event $\{C = 32\}$ *not* occurring. Observing a red ball is represented by the event $\{C = 32\}$ occurring, or equivalently by event $\{C = 16\}$ not occurring. Recall that an event is an FRP with values 0 or 1, so $\{C = 16\}$ occurring means that the FRP $\{C = 16\}$ has the value 1.

Our knowledge of B having observed a blue ball is described by the Kind $\text{kind}(B) \mid C = 16$, where we have applied a conditional constraint. The chosen

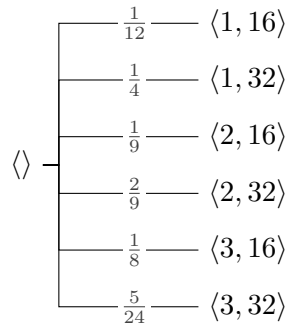
box given this observation is represented by an FRP that we write as $B \mid C = 16$, and $\text{kind}(B \mid C = 16) = \text{kind}(B) \mid C = 16$.

If we have B alone or C alone, we cannot find $B \mid C = 16$ because that depends on the values of *both* FRPs B and $\{C = 16\}$. So to find its Kind, we must work with an FRP that determines both the value of B and whether the given event has occurred. We must start with $B \triangleright S$.

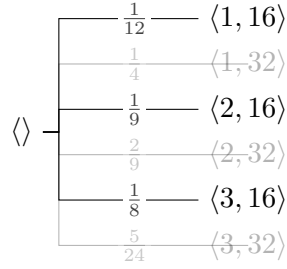
Specifically, to find $\text{kind}(B) \mid C = 16$, we start with $\text{kind}(B \triangleright S)$.

```
pgd> b = uniform(1, 2, 3)
pgd> s = conditional_kind({
...>   1: either(16, 32, '1/3'),
...>   2: either(16, 32, '2/4'),
...>   3: either(16, 32, '3/5'),
...> })
pgd> B = frp(b)
pgd> S = conditional_frp(s)
pgd> BC = B >> S
pgd> kind(BC)      # Equal to b >> s
```

which yields



The FRP BC and its Kind describe the values of B and C *jointly*, so we can determine which possibilities are or are not consistent with the conditional constraint. We apply the conditional constraint by eliminating the branches that are inconsistent with $C = 16$.



to produce

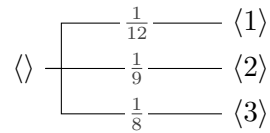
$$\begin{array}{c}
\langle \rangle \text{ --- } \frac{1}{12} \text{ --- } \langle 1, 16 \rangle \\
\text{--- } \frac{1}{9} \text{ --- } \langle 2, 16 \rangle \\
\text{--- } \frac{1}{8} \text{ --- } \langle 3, 16 \rangle
\end{array} \tag{5.3}$$

In the playground, we get this by

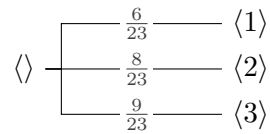
```
pgd> kind(BC) | (Proj[2] == 16)
```

where the conditional constraint is specified by applying the statistic to the right of the “given” bar to the values of the Kind or FRP to the left of the bar.

Notice that (i) applying the conditional constraint has not changed the weights of the remaining branches, and (ii) all the branches have values where $C = 16$ because of the constraint. We are interested in the Kind of B , so we transform the Kind (5.3) by projecting onto the first component



Also, it is optional but helpful to follow our standard practice and convert this Kind to canonical form, which yields $\text{kind}(B) \mid C = 16$:



We get this in the playground with any of

```

pgd> (kind(BC) | (Proj[2] == 16))[1]
pgd> Proj[1](kind(BC) | (Proj[2] == 16))
pgd> (kind(BC) | (Proj[2] == 16)) ^ Proj[1]
pgd> Proj[1] @ kind(BC) | (Proj[2] == 16)

```

Pay attention to the parentheses here. The form of these and the new @ operator will be explained below.

The playground only uses the basic form $K \mid \zeta$ and $X \mid \zeta$ for applying conditional constraints. So, if we want to compute a more general form, such as $\text{kind}(B) \mid C = 16$ in the previous example, we need to do the translation of equation (5.2) manually. The first three lines above

```

(kind(BC) | (Proj[2] == 16))[1]
Proj[1](kind(BC) | (Proj[2] == 16))
(kind(BC) | (Proj[2] == 16)) ^ Proj[1]

```

are variations on that, where we compute the joint Kind with the conditional constraint and then transform it with proj_1 . The fourth line provides a shortcut

```
Proj[1] @ kind(BC) | (Proj[2] == 16)
```

using the @ operator. If **stat** is a statistic and **U** is a Kind, then **stat @ U** is the same as **stat(U)** or $U \wedge \text{stat}$, with two exceptions: @ has higher precedence than \wedge or \mid and the result “remembers” that it came from **U** in conditional constraints. So:

```

pgd> kind_B = Proj[1] @ kind(BC)
pgd> kind_B | (Proj[2] == 16)

```

produces $\text{kind}(B) \mid C = 16$, where **kind_B** looks like the $\text{kind}(B)$ but with some extra information behind the scenes. If we try instead to do just $\text{kind(BC)}[1] \mid (\text{Proj}[2] == 16)$, we will get an error because $\text{kind(BC)}[1]$ does not have that context.

Example 5.3 Points in a Circle, With Constraints In Example 4.8, we built a two-dimensional FRP, call it P here, that represents a random point with integer coordinates within a circle of radius 5, where all valid points have equal weight. Express P in terms of its component FRPs, $P = \langle X, Y \rangle$, where X and Y represent the x - and y -coordinates of the random point.

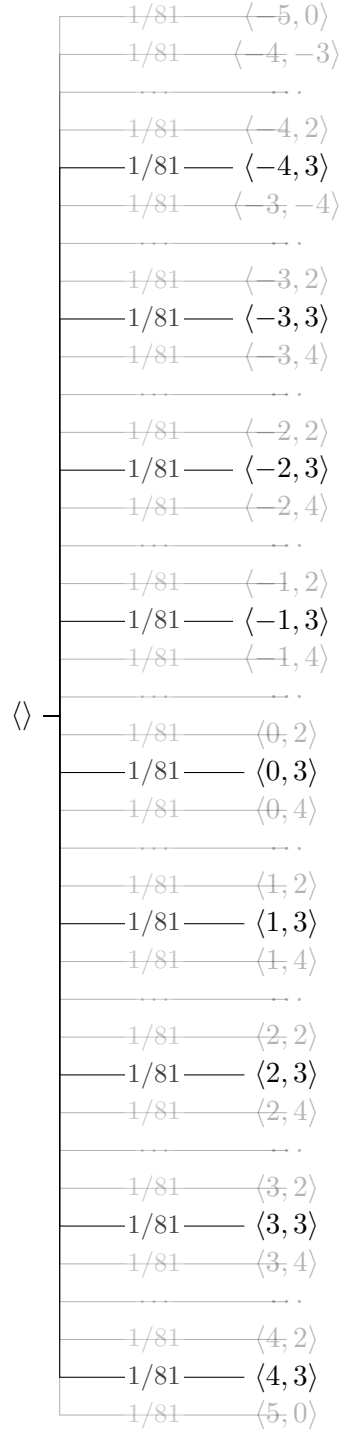
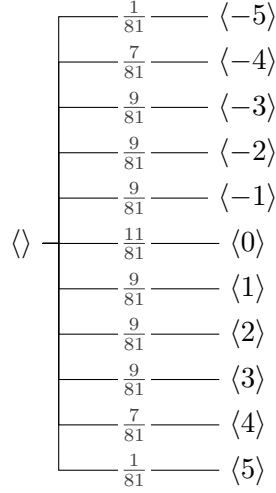


FIGURE 40. The Kind of $P \mid Y = 3$ showing the eliminated branches, in Example 5.3.

The Kind of X embodies all our predictions about the x -coordinate alone, and the Kind of Y embodies all our predictions about the y -coordinate alone. Both of these have *the same kind* (i.e., $\text{kind}(\text{proj}_1(P)) = \text{kind}(\text{proj}_2(P))$)



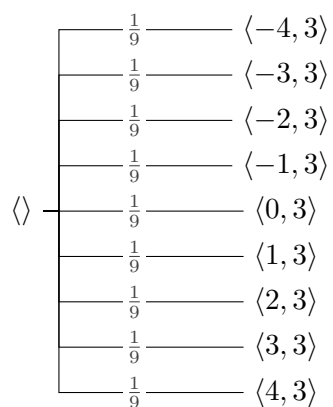
because the points in the circle are symmetric under a 90-degree rotation. Without any information about the other coordinate, any point in the circle is possible, and the weight on each value of the coordinate is proportional to the number of points with that coordinate (e.g., there are 11 points with x -coordinate 0, with y from -5 to 5).

With information about one coordinate, however, our predictions about the other coordinate *change*. Suppose, for example, we observe that $Y = 3$. Remember $\{Y = 3\}$ is an *event* – an FRP that has values 0 and 1. To say that we observe that $Y = 3$ means that we can take it has a fact that the event $\{Y = 3\}$ has occurred. So the event FRP has a value 1 and the FRP Y has value 3.

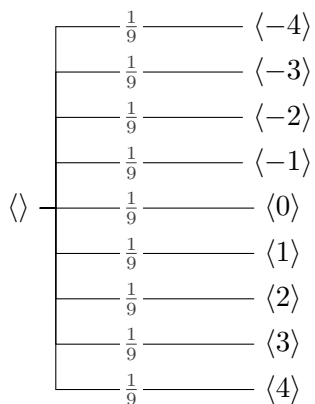
If we know that $Y = 3$, then P must be a point of the form $\langle x, 3 \rangle$ with x an integer with $x^2 \leq 25 - 9 = 16$, i.e., $x \in [-4..4]$. So, if we eliminate all the branches that are inconsistent with this partial information, we get the updated Kind for $P \mid Y = 3$ in Figure 40 where all the inconsistent branches have been crossed out. Where before we had equal weight on each of 81 points, we now have 9 remaining branches all with the same weight. An important observation:

when we eliminate branches due to the conditional constraint, **the relative sizes of the weights do not change** on the remaining branches.

When we renormalize to canonical form, the Kind of $P \mid Y = 3$ becomes



And we get the Kind of $X \mid Y = 3$ by transforming this with the statistic proj_1 , yielding the Kind



Compare this to the Kind of X without any information about Y , shown above. Once we know the value of $Y = y$, X must be one of the values in the circle along the horizontal line of height y , and all of those values are equally likely.

In the playground, using `circle_points` from the earlier example, try

```
pgd> P = circle_points()
pgd> kind(P)
```

```
pgd> kind(P) | (Proj[2] == 3)
pgd> Proj[1] @ kind(P) | (Proj[2] == 3)
```

The output is omitted but should match the displays above. Try it!

Example 5.4 Jockeying for a Win

The *Uncertain Stakes* is one of the most exclusive horse races on the world circuit but little noted by the general public. This year's race has eight elite contenders, numbered as follows

1. *Aldous Aboard*
2. *Eggs Billingsley*
3. *Cinlar's Challenge*
4. *Feller Beast*
5. *Kissing Kolmogorov*
6. *Levy Leaving*
7. *Markov Mania*
8. *Pitman's Pride*

These horses vary strongly in their response to the track conditions. Some like the track muddy; some like it dry; some do better in driving rain. There are many questions we might ask, but consider two. *Who will win? If we observe that Cinlar's Challenge wins, what can we say about the track conditions?*

Let T be the FRP representing the track conditions and W is the conditional FRP of the winning horse given the track conditions. T 's values 0, 1, 2, 3 represent the conditions fast (dry, even resilient surface), muddy (wet without standing water), sloppy (saturated with water, with standing water), and slow (wet on both surface and base layers). For each track condition, W has values 1–8 corresponding to the number of the winning horse.

Based on the horses' histories, we can assume that $\text{kind}(W)$ is the conditional Kind shown in Figure 41. We can see, for instance, that fast conditions favor *Aldous Aboard* and *Eggs Billingsley*, sloppy condition are dominated by *Feller Beast*, and slow conditions disadvantage *Levy Leaving*.

Define FRPs $R = T \triangleright W$ and $H = \text{proj}_2(R)$. The former, a two-dimensional FRP, represents both track conditions and the winning horse, and the latter

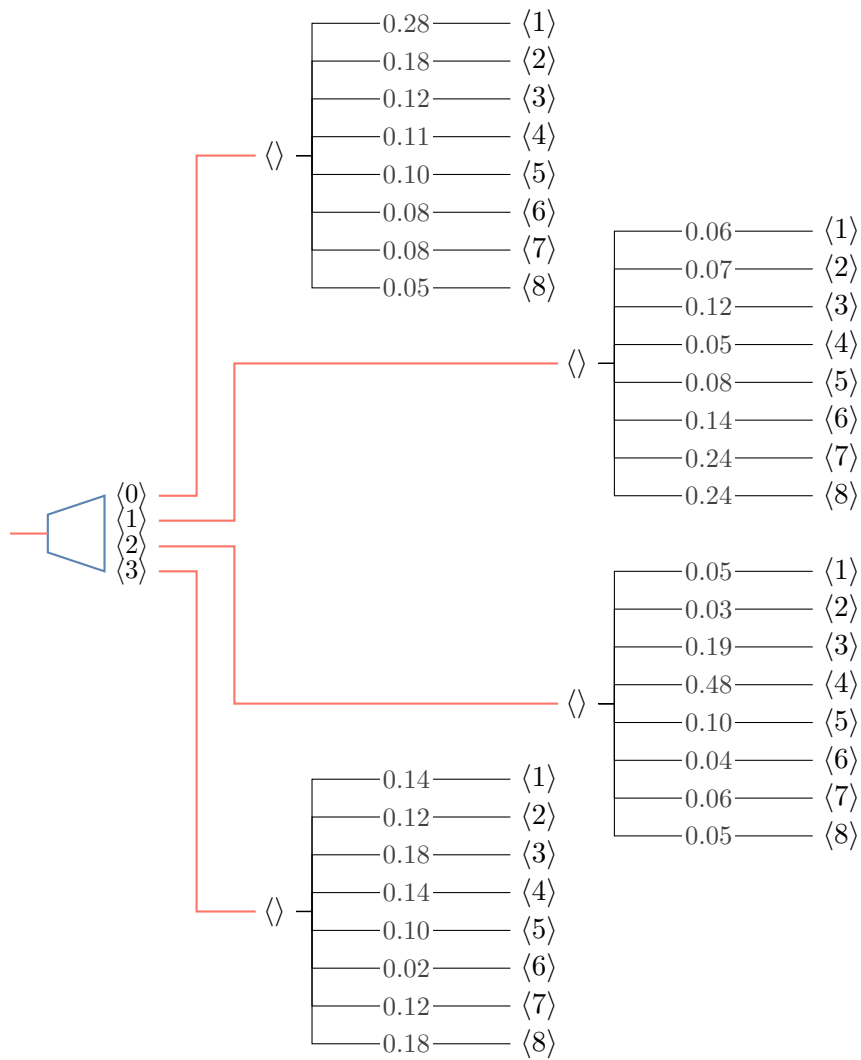


FIGURE 41. The conditional Kind of the winning horse given track condition, $\text{kind}(W)$, in Example 5.4. The inputs represent a fast (0), muddy (1), sloppy (2), and slow (3) track. The value of the Kinds is the winning horse's number.

represents the winning horses, whose number is the second component of the mixture R . Observe that $T = \text{proj}_1(R)$, so T and H are R 's component FRPs.

We are interested in (i) predicting who will win this year's *Uncertain Stakes*, both with and without information about the track conditions, and (ii) *inferring* the track conditions *given* that *Cinlar's Challenge* is observed to win the race. In the first case, we want to find $\text{kind}(H)$ and with information that the track has condition c , $\text{kind}(H) \mid T = c$. In the second case, we apply a conditional constraint on H to find $\text{kind}(T) \mid H = 3$. Note that in both cases, the conditional constraint can come from real partial information that we observe or from *counterfactual* partial information that we are interested in considering. In the latter, our questions sound like "What would we predict about which horse wins if we knew the track had condition c ?" or "What would we infer about the track condition if we knew that *Cinlar's Challenge* wins?"

You can load this example in the playground by entering

```
pgd> from frplib.examples.horse_race import T, W
```

Then do

```
pgd> R = T >> W
```

```
pgd> H = R[2]
```

Look at the Kinds of these FRPs, including the assumed Kind of T , which was not shown above. Indeed, we can immediately address the first question with

```
pgd> kind(H)
,---- 0.10200 ----- 1
|---- 0.080000 ---- 2
|---- 0.16600 ----- 3
|---- 0.25500 ----- 4
<> -|
|---- 0.096000 ---- 5
|---- 0.058000 ---- 6
|---- 0.11600 ----- 7
`---- 0.12700 ----- 8
```

which gives the chance of winning for each of the horses, absent any other information. Make sure you understand where this comes from. A good place to start would be to look at `unfold(kind(R))` and apply our procedure for computing a mixture of Kinds to compute `kind(R)`. Then apply `Proj[2]`.

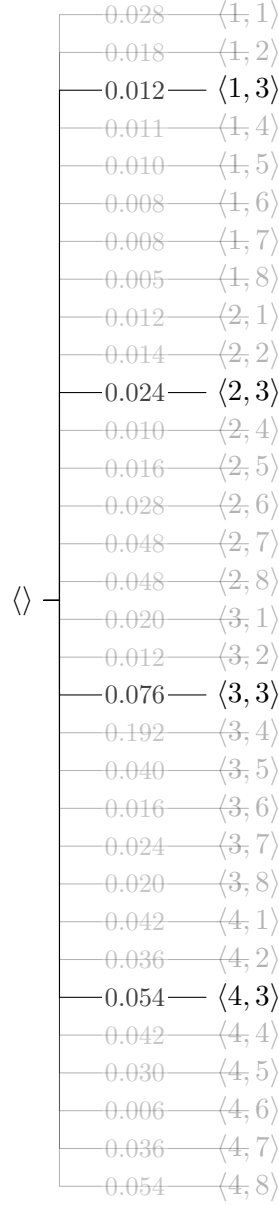
Our predictions change with additional information. For instance, if we observe that the track is muddy, the Kind of the winning horse is instead

```
pgd> Proj[2] @ kind(R) | (Proj[1] == 1)
,---- 0.060000 ---- 1
|---- 0.070000 ---- 2
|---- 0.120000 ----- 3
|---- 0.050000 ---- 4
<> -|
|---- 0.080000 ---- 5
|---- 0.140000 ----- 6
|---- 0.240000 ----- 7
`---- 0.240000 ----- 8
```

This is $\text{kind}(H) \mid T = 1$ and can also be obtained from $\text{kind}(W)$.

For the second question, we apply a conditional constraint that *Cinlar's Challenge* wins, meaning that the event $\{H = 3\}$ occurs. We want the Kind of T given that constraint: $\text{kind}(T) \mid H = 3$. We find this in three steps:

1. Start with a Kind that represents an FRP that determines *both* the value of T and whether the event occurred.
2. Eliminate all the branches in the Kind of R that are inconsistent with the given event, i.e., for which $H \neq 3$. All other branches remain *as is*.
3. Apply a projection statistic to this Kind that extracts the value of T , and optionally, convert the resulting Kind to canonical form if desired.



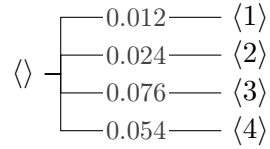
The Kind of R showing branches eliminated by the constraint $\{H = 3\}$.

Let's see these steps in action:

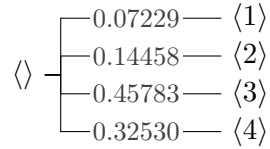
1. We start the Kind of R , which has T and H as components. If we know the value of R we can find the value of T and the value of H and can therefore

determine whether $H = 3$.

2. Eliminating branches with $H \neq 3$ from the Kind of R looks like the Kind shown above, with the eliminated branches grayed out.
3. Applying proj_1 to the Kind in step 2 yields



and following our standard practice, we convert this to canonical form



where the weights are respectively $6/83, 12/83, 38/83, 27/83$.

In the playground, the three steps are direct:

```
pgd> kind(R)
pgd> kind(R) | (Proj[2] == 3)
pgd> Proj[1] @ kind(R) | (Proj[2] == 3)
```

and the output should agree with the above displays.

The key takeaway about conditional constraints is: **to account in our analysis for partial information about the value of an FRP – observed or hypothetical – we eliminate from consideration any possible values that are inconsistent with the information.** The weights on the remaining branches do not change, though we may renormalize them into canonical form.

Keep in mind that conditions can be combined using logical operators: logical-and (operator \wedge), logical-or (operator \vee), and occasionally logical “not” (operator $!$). These operators tend to be convenient when writing complicated conditions, but it’s fine to use words (**and**, **or**, **not**) instead in your work. So, $\{\text{proj}_2(A) = 1 \wedge \text{proj}_1(A) = 1\}$ is the event that *both* of A ’s first two components are one, $\{\text{proj}_2(A) = 1 \vee \text{proj}_1(A) = 1\}$ is the event that *at least one* of A ’s first two components are one. Note that for multiple dimensional FRPs like A , we can use subscripts to denote components, so $A_1 = \text{proj}_1(A)$ and $A_2 = \text{proj}_2(A)$. In the playground, things are a little more awkward because of restrictions imposed by Python. We can use the combinators **And**,

Or, and Not for logical and, or, and not. The first two of these take any number of conditions and can be nested to produce arbitrary Boolean expressions. For example,

```
A | And(Proj[2] == 1, Proj[1] == 1)
A | Or(Proj[2] == 1, Proj[1] == 1)
A | Not( And(Proj[2] == 1, Proj[1] == 1) )
```

and so forth. And in the playground, we can use bracketed indexing on an FRP or Kind as a short hand for projection, so `A[2]` is the same as `Proj[2](A)` or `A ~ Proj[2]`.

As we saw in the previous examples, a common situation is that we describe a multi-stage process with a mixture, observe the value of a later stage, and use that observation as a conditional constraint to *infer* the value of an earlier stage. So for instance, we see the color of the selected ball and infer the chosen box it came from or see the winner of the race and infer the track conditions. Suppose A is an FRP of dimension m and M a conditional FRP of type $m \rightarrow n$. Let $X = A \triangleright M$ and $B = \text{proj}_{(m+1)..}(X)$, where $A = \text{proj}_{..m}(X)$.⁵⁷ If we observe that B has value b , what do we learn about A ? The answer: $\text{kind}(A) \mid B = b$. By equation (5.2), we find this in three steps:

1. Find the joint Kind using the mixture, $\text{kind}(X) = \text{kind}(A \triangleright M)$.
2. Apply the conditional constraint $\{B = b\}$ to this Kind, $\text{kind}(A \triangleright M) \mid B = b$.
3. Project onto the components describing A , $\text{proj}_{..m}(\text{kind}(A \triangleright M) \mid B = b)$.

In the playground, we can write this simply as

```
pgd> Proj[: (m+1)] @ kind(A >> M) | (Proj[(m+1):] == b)
```

where the three steps correspond to

1. `kind(A >> M)`, the Kind of both stages jointly
2. `kind(A >> M) | (Proj[(m+1):] == b)`, applying the conditional constraint, and
3. `Proj[: (m+1)] @ kind(A >> M) | (Proj[(m+1):] == b)`, projecting onto the components of interest.

(Recall that slices `i:j` in Python do not include the final index `j`.)

Taken together, these steps form an operation called **Bayes's Rule** in which we **infer earlier components of a mixture from the observations of later components**. The function `bayes(observed_y, x, y_given_x)` carries out this

⁵⁷Recall that $\text{proj}_{i..j}$, $\text{proj}_{..j}$, and $\text{proj}_{i..}$ are respectively equivalent to $\text{proj}_{i,i+1,\dots,j}$, $\text{proj}_{1,2,\dots,j}$, and $\text{proj}_{i,i+1,\dots}$.

operation in the playground, where `observed_y` is the observed value of a quantity, `x` is the Kind of another quantity, and `y_given_x` is the conditional Kind of the quantity `y` *given* a value of `x`. The result is the Kind⁵⁸ of the value of `x` with the conditional constraint that `y` is the observed value.

⁵⁸bayes works with FRPs as well but we almost always use it with Kinds.

Let see this in action by revisiting Example 4.5 about disease testing. We know the prevalence of a disease in the population (1/1000), the sensitivity of the test (ability to correctly detect someone with the disease, 950/1000), and the specificity of the test (ability to correctly determine someone does not have the disease, 990/1000). We specify that information in the playground as follows.

```
pgd> has_disease = either(0, 1, 999)      # No disease has higher weight
pgd> test_by_status = conditional_kind({
...>   0: either(0, 1, 99), # No disease,  negative higher weight
...>   1: either(0, 1, 1/19) # Yes disease, positive higher weight
...> })
pgd> dStatus_and_tResult = has_disease >> test_by_status
pgd> dStatus_and_tResult

,---- 98901/100000 ---- <0, 0>
|---- 999/100000 ---- <0, 1>
<> -|
|---- 1/20000 ---- <1, 0>
`---- 19/20000 ---- <1, 1>

pgd> Disease_Status = Proj[1]      # Naming this statistic
pgd> Test_Result = Proj[2]        # ...and this statistic
```

This produces a Kind with two components that we name to aid undersanding. Our question is: if someone tests positive how likely are they to have the disease. Think for a moment about how you can do this in the playground. Given the value of the `Test_Result` component, what can we infer about the `Disease_Status` component?

Puzzle 42. Try to craft a single expression in the playground to answer our main question: if someone tests positive how likely are they to have the disease.

We can answer our question with a conditional constraint on the observed information of test result and a projection onto disease status.

```
pgd> Disease_Status @ dStatus_and_tResult | (Test_Result == 1)
,---- 999/1094 ---- 0    # No disease
<> -|
`---- 95/1094 ---- 1    # Yes disease
```

We restrict attention to values for which the test is positive (our condition) and then marginalize to look only at disease status. That's it. We can rewrite this in terms of the `bayes` function with the observed value of the test as the first argument:

```
pgd> bayes(1, has_disease, test_by_status)
,---- 999/1094 ---- 0    # No disease
<> -|
`---- 95/1094 ---- 1    # Yes disease
```

The result may be surprising: despite a positive test, the probability that the patient has the disease is small. The small weight on having the disease even with a positive test result derives from the low baseline prevalence of the disease in the population, i.e., the small weight on $\langle 1 \rangle$ in the Kind `has_disease`. Work out carefully how this result was derived. The amazing thing is how simple it is; we just exclude branches and renormalize. The statistic `Disease_Status` selects one component, and given that the test result is *known* the other component is not even that interesting. (Take a look at the tree without the marginalizing projection to see this.)

Here, `test_by_status` is a *conditional Kind* because it specifies a Kind that is contingent on some other specified value, the patient's disease status. The uses of the word “conditional” in “conditional Kind” and “conditional constraint” are directly connected. For instance, when you evaluate

```
dStatus_and_tResult | (Disease_Status == 0)
dStatus_and_tResult | (Disease_Status == 1)
```

you will see that these are just `test_by_status(0)` and `test_by_status(1)`, respectively. The conditional Kind gives for each input the Kind obtained by applying a conditional constraint that that input was observed! Notice also that

```
pgd> Kind.equal( Disease_Status(dStatus_and_tResult), has_disease )
True
```

since `has_disease` is just the top level of the unfolded Kind `dStatus_and_tResult`.

In general, if k is a Kind of dimension d_k and m is a conditional Kind of type dimension $d_k \rightarrow d_k + d_m$, then the Kind $k \triangleright m$ has dimension $d_k + d_m$. From any *value* of $k \triangleright m$, we can extract the k value with `Proj[: (d_k+1)]` and the corresponding m value with `Proj[(d_k+1):]`. Define

```
pgd> ks_values = Proj[: (d_k+1)]
pgd> ms_values = Proj[(d_k+1):]
```

Then for every value v of k :

```
Kind.equal( k, ks_values(k >> m) )
Kind.equal( m(v), ms_values @ k >> m | (Proj[ks_values] == v) )
```

are both `True`. In other words, for the conditional Kind m , $m(v)$ is the *Kind given the condition* that k 's value equals v . So m just packages all the conditionals given each value of k . Mathematically, we can state this precisely in the same way.

Suppose k is a Kind of dimension d_k and m is a compatible conditional Kind of type $d_k \rightarrow d_k + d_m$, then

$$k = \text{proj}_{1..d_k}(k \triangleright m) \quad (5.4)$$

and for every value v of k ,

$$m(v) = \text{proj}_{(d_k+1)..}(k \triangleright m \mid \text{proj}_{1..d_k}(\blacksquare) = v). \quad (5.5)$$

Recall that `Proj[a: (b+1)]` is a statistic that extracts components $a, a+1, \dots, b$. In math we write this as $\text{proj}_{a..b}$.

Checkpoints

After reading this section you should be able to:

- Explain in broad terms how we account for partial information about the value of an FRP, observed or hypothetical.
- Define a condition and construct several examples, mathematically and in the playground.
- Show how to combine conditions with logical-and and logical-or (and logical-not).
- Define an event and explain what it means for an event to occur or not occur.
- Show how to convert a Boolean expression to an event and how to denote the corresponding FRP.
- Write a Kind or FRP with a conditional constraint.
- Explain what an *FRP given a condition* and a *Kind given a condition* mean.
- Describe how to find $K \mid \zeta$ in canonical form when you are handed a Kind K in canonical form and a compatible condition ζ .
- Identify the difference, if any, between $K \mid \top$ and K for a Kind K .
- Explain the purpose of Bayes's Rule, and describe the steps that comprise it.
- Apply Bayes's Rule in the playground.

6 Three Dialogues: Computation, Systems, Simulation

Key Take Aways

In friendly interactions with clients and employees of the FRP Warehouse, you explore practical aspects of building probabilistic systems to solve problems.

A Dialogue on Computation highlights several computational techniques that are frequently useful, including “monoidal” parallelism, symmetry, and sampling.

A Dialogue on Systems and State examines how to build random systems that evolve over time or space and describes the idea of a system’s *state*.

A Dialogue on Solutions and Simulation develops tools for computing answers to questions about the long-term evolution of random systems as solutions to “one-step” equations or as simulations of the system’s dynamics.

6.1 A Dialogue on Computation

Alice and Bob are clients of the FRP Warehouse whom you met during the orientation for new users. This conversation took place during an orientation workshop.

BOB: I’m getting frustrated, Alice. This calculation is hanging.

ALICE: The dice example again? What’s the problem?

BOB: I want to compute the Kind for an FRP that models the sum of 100 rolls of a six-sided dice. So I define the Kind for one roll, `d6 = uniform(1, 2, 3, 4, 5, 6)`, compute the Kind for 100 rolls – `d6 ** 100` – and then transform ...

ALICE: Well that’s your problem right there. What is the size of `d6 ** 100`?

BOB: There are 6 possibilities for each of 100 rolls, so 6^{100} possible values. Ah...that’s a big number. No wonder it’s taking so long.

But the sum of the rolls doesn’t care about the distinction between most of those possibilities, so it seems it should be possible to do this calculation efficiently.

ALICE: It actually is. I’ve been considering that problem for another project. What does the playground display when you print the statistic `Sum`?

BOB: Let’s see. It says

A Monoidal Statistic ‘sum’ that returns the sum of all the components of the given value. It expects a tuple and returns a scalar.

What the heck is a “Monoidal Statistic?”

ALICE: That threw me too, but after I dug into it, I realized it was a simple idea.

The `Sum` statistic takes in a value that is a list of numbers and adds up all the components to give a number, so it takes in a list of numbers and returns a number. What happens to the sum if you add some elements to the list, as we do when we take mixtures?

BOB: The `Sum` just adds up the new elements and adds that sum to the total.

ALICE: Exactly! Let’s write `::` for the operation of joining two lists, so $\langle 10, 20, 30 \rangle :: \langle 40, 50 \rangle = \langle 10, 20, 30, 40, 50 \rangle$ and $\langle 10, 20 \rangle :: \langle \rangle = \langle 10, 20 \rangle$ and so on. What you said is

$$\begin{aligned} \text{Sum}(\langle 10, 20, 30 \rangle :: \langle 40, 50 \rangle) &= \text{Sum}(\langle 10, 20, 30 \rangle) + \text{Sum}(\langle 40, 50 \rangle) \\ &= \text{Sum}(\langle \text{Sum}(\langle 10, 20, 30 \rangle), \text{Sum}(\langle 40, 50 \rangle) \rangle), \\ \text{Sum}(\langle 10, 20 \rangle :: \langle \rangle) &= \text{Sum}(\langle 10, 20 \rangle) + \text{Sum}(\langle \rangle) \\ &= \text{Sum}(\langle \text{Sum}(\langle 10, 20 \rangle), \text{Sum}(\langle \rangle) \rangle), \end{aligned}$$

because the sum of an empty list is 0. Make sense?

BOB: That’s a mouthful, but yes, I see. $\text{Sum}(a :: b) = \text{Sum}(\langle \text{Sum}(a), \text{Sum}(b) \rangle)$. So I can apply `Sum` as I go along and get the same answer. That means that the Kind `Sum(d6 ** 100)` is equal what I get by doing

```
pgd> sum_of_4_rolls = Sum(d6 ** 4)
pgd> sum_of_100_rolls = sum_of_4_rolls      # initialize
pgd> for _ in range(24):                    # loop to successively update
...>     sum_of_100_rolls = Sum(sum_of_100_rolls * sum_of_4_rolls)
```

ALICE: I think so, but let’s do a simpler example to make sure we understand it correctly. Suppose we are just summing 12 rolls. The values of `d6 ** 4` are lists with four numbers in $[1..6]$ like $\langle 1, 4, 3, 5 \rangle$, $\langle 3, 6, 5, 6 \rangle$, and $\langle 6, 2, 1, 1 \rangle$. Transforming by `Sum` adds these up giving values for `Sum(d6 ** 4)` like $\langle 13 \rangle$, $\langle 20 \rangle$, and $\langle 10 \rangle$ respectively. Your `sum_of_4_rolls` looks like

```
,---- 1/1296    ---- 4
|---- 4/1296    ---- 5
|---- 10/1296   ---- 6
```

```

|---- 20/1296 ---- 7
|---- 35/1296 ---- 8
|---- 56/1296 ---- 9
|---- 80/1296 ---- 10
|---- 104/1296 ---- 11
|---- 125/1296 ---- 12
|---- 140/1296 ---- 13
<> -+---- 146/1296 ---- 14
|---- 140/1296 ---- 15
|---- 125/1296 ---- 16
|---- 104/1296 ---- 17
|---- 80/1296 ---- 18
|---- 56/1296 ---- 19
|---- 35/1296 ---- 20
|---- 20/1296 ---- 21
|---- 10/1296 ---- 22
|---- 4/1296 ---- 23
`---- 1/1296 ---- 24

```

If we mix it with itself, `sum_of_4_rolls * sum_of_4_rolls` corresponds to rolling 4 dice once and then rolling 4 dice again and recording a pair of sums, with values like $\langle 13, 8 \rangle$ and so on. Then, `Sum(sum_of_4_rolls * sum_of_4_rolls)` adds those values up, giving us the sum of eight dice. And doing this yet again gives us

```
Sum(sum_of_4_rolls * sum_of_4_rolls * sum_of_4_rolls)
```

which is like rolling 4 dice three times, getting the sums for each set of 4, and then adding up those subtotals to get the total sum. This is the Kind of the sum of 12 rolls as we wanted and is the same as:

```
Sum( Sum(d6 ** 4) * Sum(d6 ** 4) * Sum(d6 ** 4) )
```

BOB: Excellent. So “monoidal statistics” like `Sum` are those that let you do this decomposition and compute the statistic in parallel. They could have called them “parallel statistics,” eh?

Looking at the predefined statistics in the playground, I see that `Min`, `Max`, and `Count` also have this property. I suppose that makes sense; after all,

$$\begin{aligned}\min(\langle 10, 20, 30 \rangle :: \langle 40, 50 \rangle) &= \text{Min}(\langle \text{Min}(\langle 10, 20, 30 \rangle), \text{Min}(\langle 40, 50 \rangle) \rangle) \\ \min(\langle 10, 20 \rangle :: \langle \rangle) &= \text{Min}(\langle \text{Min}(\langle 10, 20 \rangle), \text{Min}(\langle \rangle) \rangle),\end{aligned}$$

which looks just like the formula for `Sum` above. (We take the minimum of an empty list of numbers to be ∞ by convention.)

ALICE: Right, so we have basically the same formula $\text{Min}(a :: b) = \text{Min}(\langle \text{Min}(a), \text{Min}(b) \rangle)$.

BOB: So, we can get the Kind for the minimum of 12 rolls by

```
Min( Min(d6 ** 4) * Min(d6 ** 4) * Min(d6 ** 4) )
```

like before. That’s great, but what if I want to do something more complicated, like the mean of the rolls or the range (difference between max and min). Those statistics don’t have this property.

ALICE: True, but we can get them both from statistics that do. For instance, if you can find the Kind of the sum, you can transform that to get the mean with `sum_of_100_rolls ^ (_ / 100)`.

But let’s solve the problem more generally. Have you seen the `Fork` combinator in the playground?

BOB: Yes, it combines a bunch of statistics with common dimension into a big tuple containing all of their results. For example, $\text{Fork}(s_1, s_2)(x) = s_1(x) :: s_2(x)$ and $\text{Fork}(s_1, s_2, s_3)(x) = s_1(x) :: s_2(x) :: s_3(x)$.

ALICE: And notice that if the statistics you give to `Fork` are “monoidal statistics”, so is the statistic that it returns.

BOB: Because we can just apply our formula above to each component.

ALICE: Yes. So if you want to compute the range (max - min), apply our formula above with the statistic `min_max = Fork(Min, Max)` and then take the difference at the end. That is,

```
min_max(min_max(d6 ** 4) * min_max(d6 ** 4) * min_max(d6 ** 4)) ^ Diff
```

BOB: Beautiful! Complicated but beautiful.

ALICE: Yes, it's a lot. The good news is that the playground can automate this in many cases, but that's a story for another day.

BOB: My problem is solved, thanks.

ALICE: Well, I have a related problem. You know how much I enjoy playing poker.

BOB: You're a shark!

ALICE: Well, I thought I might parlay that interest into a way to beat the Warehouse. I'm trying to create FRPs to model shuffling a deck of cards, by drawing one card at a time.

BOB: I see. The next card depends on which cards you've seen already. Sounds like a mixture.

ALICE: Exactly, but I found it a bit tricky to define. Can I show you? Fair warning, there's some Python here.

BOB: I'm not really fluent in Python, but I'm guessing I can follow along.

ALICE: Absolutely you can, it should be clear, though I'll explain any Python oddities. Let's start with a standard deck of 52 cards. We'll arbitrarily assign the cards numbers 1 through 52; we can be more specific later if needed. At the first stage, I need an FRP that selects each card with equal weight; that's just

```
pgd> card1 = uniform(1, 2, ..., 52)
```

For the next card, I need a conditional Kind that picks uniformly among *all but the first card chosen*. I'll use the playground function `irange` that gives an *inclusive* range of integers from its first to second arguments, with an option to exclude values in a set. This looks like

```
pgd> card2 = conditional_kind({
...>   (first_card,): uniform( irange(1,52, exclude={first_card}) )
...>   for first_card in irange(1,52)
...> })
```

For each card, an integer from 1..52, this uses the `uniform` factory to make an equally weighted kind on all the other cards.

BOB: And your code is building a mapping of key-value pairs for each value of

`first_card`, where `(first_card,)` is the key and the Kind excluding `first_card` is the value.

ALICE: Right. A conditional Kind maps the values of one Kind to other Kinds of equal dimension. Now, I could continue like this all the way to `card52`, but I think I need to be more systematic. Here's what I tried, a function that returns a conditional Kind for a particular card draw:

```
def card(n):
    "Returns the conditional kind for the nth card drawn."
    if n == 1:
        return uniform(1, 2, ..., 52) # (1)

    def draw_kind(previous_cards): # (2)
        next_cards = list( irange(1, 52, exclude=set(previous_cards)) ) # (3)
        return uniform(next_cards) # (4)

    return conditional_kind(draw_kind) # (5)
```

In (1), if this is the first card, we need to start things off with no previous cards, so we return a conditional Kind of type $0 \rightarrow 1$, which is just an ordinary Kind. In (2), we receive the list of $n - 1$ and compute a Kind for the next card, so this is a conditional Kind of type $n - 1 \rightarrow n$ that we define as a Python function. In (3), we create the list of valid next cards, which just excludes all the previous cards. In (4), we use the `uniform` factory to produce a Kind with equal weight on all of these cards. And finally, in (5) we convert the function `draw_kind` to a conditional Kind object, using `conditional_kind`.

BOB: OK, there's some hairy stuff there, but I'm generally following. How do you use this?

ALICE: Well `card(1)` is the Kind after one drawn card, and in succession

```
pgd> card(1) >> card(2)
pgd> card(1) >> card(2) >> card(3)
pgd> card(1) >> card(2) >> card(3) >> card(4)
```

give the Kind of the shuffle after 2, 3, and 4 cards are picked, respectively. We could write a loop to do it for all cards

```
pgd> shuffle = card(1)
pgd> for n in irange(2, 52):
...>     shuffle = shuffle >> card(n)
```

Of course, that's impossibly slow because there are 52! different shuffles in the tree, another big number.

BOB: So, you're looking for a trick like what worked for my problem.

ALICE: A trick would be nice, but I want to *understand* my options for tackling this.

BOB: I have two thoughts. First, what questions are you trying to answer? In my case, it mattered that I was interested in the Sum, for example, which made it possible to reduce the complexity. If you want to predict your hand, say, then you don't need to draw all 52 cards.

Second, are you sure you need an exact answer?

ALICE: It's true that if I'm looking at what happens in my hand it's easier. Like if I've drawn five specific cards, and I want to know the chance of getting a fifth card, it's easier, but I still may have to deal with 20, 25, 30 cards.

BOB: What happens if you permute the labels? Does it matter if you observe cards 1, 5, 9, 13, and 17 (in a four player game with five cards each) versus cards 1, 2, 3, 4, and 5?

ALICE: Interesting. I think that's an important observation, and I want to come back to that. If I had cards 1-5 in my hand and were drawing the sixth, I could predict it like this, for example. Draw six cards and compute the Kind of the sixth card with the conditional constraint given the specific five cards in my hand.

```
pgd> my_hand = card(1) >> card(2) >> card(3) >> card(4) >> card(5)
pgd> next_card = (my_hand >> card(6) | (Proj[1:5] == (16,17,18,19,4))) [6]
pgd> next_card ^ 0r(__ == 20, __ == 15)
```

BOB: Cool, you're assuming you got a particular four cards in a row and want to see whether you get a straight. You could do this for any cards in your hand or write a function that checks various combinations.

ALICE: Yes, that's useful. I do want to come back to the other idea you had. You were suggesting that I demo FRPs instead of computing the Kinds?

BOB: Right. The Kinds let you compute everything exactly, but if you know what question you want to answer, you can tailor an FRP to that and demo it.

Now that I think of it, that's not a bad approach to my problem earlier.

```
pgd> d6_frp = frp(d6)
pgd> FRP.sample( 10_000, Sum(d6_frp ** 100) )
```

It's not as fast or exact as what we came up with earlier, but pretty good.

ALICE: The key is that the playground does not have to compute the Kind of an FRP like `Sum(d6_frp ** 100)` until you ask for it. It just hooks output ports to input ports and pushes the button. I could do something similar:

```
def draw(n):
    "Returns a conditional FRP for the nth card drawn."
    return conditional_frp(card(n))
```

The `conditional_frp` turns every Kind in a conditional Kind into a *new* FRP of that Kind, which is what I need.

BOB: Then just do your loop with

```
pgd> deck = draw(1)
pgd> for n in irange(2, 52):
...>     deck = deck >> draw(n)
pgd> deck
```

to get the value of a random deck, or do `FRP.sample` to demo a bunch of them. It won't be fast or exact, but it will give you useful information.

ALICE: That's quite good; I can use that. But I've also been thinking about your comment on permutations.

The `Permute` statistic factory in the playground produces statistics that just rearrange the order of the list. For example, `Permute(3,2)` swaps the second and third components in a value list.

```
pgd> psi = Permute(3,2)
pgd> psi( (10,20,30) ) = (10, 30, 20)
```

Applying a permutation to the labels for our deck is just a relabeling of the cards, but we don't really care which number is assigned to which card.

Suppose I start with some Kind k on $n - 1$ cards and compute the resulting Kind for n cards: `k >> cards(n)`. If do any permutation of the labels after this, it is equivalent to doing the *same* permutation on the values of k . That is, for any permutation "...", these two Kinds are equal

```
k >> cards(n) ^ Permute(...)      (k ^ Permute(...)) >> cards(n)
```

BOB: That's not obvious to me, but I'm trying it out in the playground and it does seem to work.

ALICE: Think of it this way. If I put new labels on *all* the cards after I've drawn $n - 1$, then since all n^{th} cards have equal weight, it's the same as if we draw the n^{th} card before doing the relabeling.

BOB: Hmm. I think I've got it. And I see where you are going. Since the Kind of the shuffled deck is

```
card(1) >> card(2) >> card(3) >> ... >> card(51) >> card(52)
```

then if we apply a permutation at the end, we can just move it up through the `>>`'s.

```
card(1) >> card(2) >> card(3) >> ... >> card(51) >> card(52) ^ Permute(...)
card(1) >> card(2) >> card(3) >> ... >> (card(51) ^ Permute(...)) >> card(52)
...
card(1) >> card(2) >> (card(3) ^ Permute(...)) >> ... >> card(51) >> card(52)
card(1) >> (card(2) ^ Permute(...)) >> card(3) >> ... >> card(51) >> card(52)
(card(1) ^ Permute(...)) >> card(2) >> card(3) >> ... >> card(51) >> card(52)
```

All these are the same Kind!

ALICE: And here's the punchline. We know that `cards(1)` is just `uniform(1,2,...,52)`, so `cards(1)` does not change when you relabel the cards.

```
Kind.equal( cards(1) ^ Permute(...), cards(1) )
```

BOB: So, permutating the deck doesn't change the Kind, or your analysis!

So, if you want to consider only your hand and a few cards to draw from, you can use `cards(1) >> ... >> cards(8)`, which is more manageable.

In fact, if the cards in your hand are `c_1`, `c_2`, `c_3`, `c_4`, and `c_5`, you can just do

```
pgd> my_hand = (c_1, c_2, c_3, c_4, c_5)
pgd> constant(my_hand) >> cards(6) >> cards(7) >> cards(8)
```

ALICE: Nice. You used the fact that `my_hand` equals the Kind `also_my_hand` where

```
pgd> first_five = cards(1) >> cards(2) >> cards(3) >> cards(4) >> cards(5)
pgd> also_my_hand = first_five | (__ == (c_1, c_2, c_3, c_4, c_5))
```

That makes it easier to answer many interesting questions. Good team work!

BOB: Don't risk too much money at the table...

ALICE: Restraint is my middle name.

BOB: (*Rolls eyes affectionately*)

Puzzle 43. Suppose you are interested in when a specific pattern of die rolls – 4, 6, 2 – occurred during successive rolls at any point during 100 rolls of a six-sided die. Using the same `d6` that Bob did in this section, compute the Kind of the *event* that the pattern occurs, i.e., an FRP that outputs 1 if the pattern occurs and 0 otherwise.

For the next puzzles, we refer to the following example.

Example 6.1. A language is a set of strings made up of symbols from a fixed alphabet. Consider the language consisting of one or more `a`'s with a zero or one commas between each sequence of `a`'s. Strings "`a,aaa,a,aaa,a`" and "`a`" and "`aaaaaa,a,a,a`" belong to this language, but strings "`a,a`" and "`,a,`" and "`,`" do not. We will describe this language by a graph whose nodes represent "states" and whose edges represent "transitions." The graph is shown in Figure 42.

We start in the blue S node – the "Start" state. We will process a string of `a`'s and `,`'s one character at a time, moving from state (node) to state (node).

Suppose we are in a particular state (node) at a given time. If the next

If you are familiar with *regular expressions*, this language is described by `a+(,a+)*`.

unseen character in the string is an **a**, we follow the edge labeled **a** out of our node. If the next unseen character in the string is an **,**, we follow the edge labeled **,** out of our node. This determines the next state

After seeing “**a,aa**”, for instance, we would be in state A; after “**a,a,**” in state C; and after “**a,,**” in state F. The green state A is “Accept” – *ending* there means that the input string belongs to the language, but ending in any other state means that it does not. The red state F is “Failure” – *reaching* that state automatically means the input does *not* belong to the language.

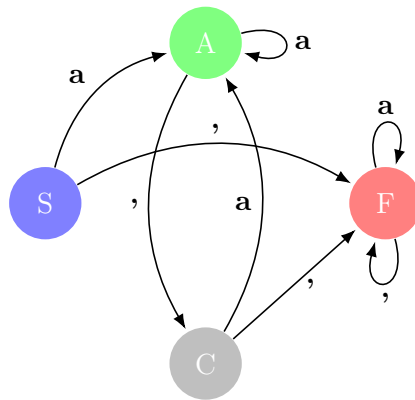


FIGURE 42. The language described in Example 6.1.

Puzzle 44. Referring to the situation in Example 6.1, assign the number 0 to the character **a** and the number 1 to the character **,** and the number 2 to an “end of string” marker, which can be repeated.

Define `char = uniform(0,1,2)`. What does `char ** 80` represent?

Write (in code or pseudo-code) a function that takes a value of `char ** 80` and returns the corresponding string.

Puzzle 45. We are interested in whether the string produced by `char ** 80` in the previous puzzle belongs to the language described by Example 6.1.

Assign the number 1 to the case where the string is accepted and 0 to the case where the string is not accepted. Compute the corresponding Kind.

You will want to construct an initial Kind and a conditional Kind for each move. Like Alice and Bob, you only need some information not the whole path.

6.2 A Dialogue on Systems and State

Carlos and Danielle work in the research division of the FRP Warehouse. This conversation took place when you met them during a tour.

CARLOS: The FRP Warehouse has a contract with the city of Uncertain, Texas to run all the local traffic lights, and our team is charged with devising new ways to build dynamic, random systems like that.

DANIELLE: And even more complicated as well.

YOU: What do traffic lights have to do with FRPs?




DANIELLE: At each tick of the clock, you can think of a traffic light as a conditional FRP. It gets input that represents the current traffic at the intersection and returns an FRP that represents how the light's state will change at the next tick of the clock.

YOU: Change *state*? Like from liquid to gas?

CARLOS: Well, not exactly, but it's a similar idea. The **state** of a system is information that fully – and if possible, concisely – describes the internal configuration of a system at any given moment.

If you know a system's state, you can describe what you will observe from the system and how the state can change.

DANIELLE: It might help to start with a simple traffic light to make this clearer.

YOU: The traffic light moves from green () to yellow () to red (), so the state of the system is the current color of the light?

DANIELLE: It can be in simple cases. However, to allow the light to behave the way we might want, it is useful to keep track of some extra information.

YOU: Ah, I see. We probably want the yellow light to be on for a shorter time than the green or red, for instance. We would need to know not only what color the light is but also how long it has been that color.

CARLOS: Exactly! We could, for instance, define the state of the traffic light to have the form $\langle c, n \rangle$ where c is the current color and n is the number of ticks of the clock for which the light has had color c . As examples, possible states are $\langle \text{red}, 0 \rangle$, $\langle \text{yellow}, 10 \rangle$, and $\langle \text{green}, 3 \rangle$.

DANIELLE: There's usually *not* just one right way to define a system's state. We *choose* how to define the state based on what we want the system to do and what

information we need to update the state.

YOU: Update the state? You mean when the clock ticks and the state is $\langle \text{red}, 7 \rangle$, the state might change to $\langle \text{green}, 8 \rangle$ or to $\langle \text{yellow}, 0 \rangle$, depending on how long we want each color to show for.

CARLOS: Yes. We can specify a rule $\langle n_g, n_y, n_r \rangle$ so the light stays red for n_g ticks of the clock, then yellow for n_y ticks, and then green for n_r ticks.

DANIELLE: We would write the rule as a *function*:

$$\langle c, n \rangle \mapsto \langle c, n + 1 \rangle \{n < n_c\} + \langle \text{next}(c), 0 \rangle \{n = n_c\}$$

where $\text{next}(\text{red}) = \text{yellow}$, $\text{next}(\text{yellow}) = \text{green}$, and $\text{next}(\text{green}) = \text{red}$.

YOU: You are using indicator functions⁵⁹ here?

⁵⁹See Section F.4.

CARLOS: Yes, we use the indicators like $\{n < n_c\}$ to select among cases. As a named Python function, this would be

```
def update_state(state):
    color, ticks = state
    if ticks < rule[color]:
        return (color, ticks + 1)
    return (next(color), 0)
```






YOU: OK, I'm still getting used to indicators, but that makes sense. I do wonder though why you need FRPs for any of this.

DANIELLE: That's a good question. One answer is that other rules are possible, and you do not always *want* a color to have the same fixed duration. You might want to randomize the light a bit so people do not learn to jump the light. You might also want to incorporate data into a guess for whether the light should stay at its current color.

CARLOS: Another answer is that this is a very simple system, and we want to build a technology for describing this *and* other more complex systems.

Let's take the next step that Danielle is suggesting and randomize the update rule we just discussed to see how this would work. You will replace the update function with a conditional FRP that takes as input the value of an FRP representing the current state.

YOU: That's clearer, thanks. Let me give it a try in the playground. (*thinks*)

DANIELLE: Excellent idea. I've loaded some infrastructure to make this easier from `frplib.examples.traffic_light`. For instance, `TrafficLight.GREEN` stands for the  color, and `next(TrafficLight.GREEN)` gives you . These colors are encoded as integers 0, 1, and 2 for , , and .

YOU: Thanks. OK, here's code I might enter into the playground:

```
change_on = {
    TrafficLight.GREEN: 1/30,
    TrafficLight.YELLOW: 1/5,
    TrafficLight.RED: 1/30,
}

@conditional_frp(codim=2, target_dim=2)
def tick_light(state):
    color, ticks = state
    change_probability = change_on[color]
    next_kind = weighted_as(
        (color, ticks + 1),
        (next(color), 0),
        weights=[1 - change_probability, change_probability]
    )
    return frp(next_kind)

start_any_color = uniform(change_on.keys()) ^ Fork(Id, 0)
```

The conditional FRP `tick_light` decides randomly at each tick whether to change the color. The dictionary `change_on` associates with each color the weight on changing color. The Kind `start_any_color` is one possible Kind for the initial state.

CARLOS: Great! Now, how would you use this to “run” the traffic light if I gave you an FRP S that represents the state of the traffic light at a particular moment.

YOU: I would compute a mixture $S \gg \text{tick_light}$, but this would give me a 4-dimensional FRP that includes the state from S and the updated state. So I would apply `Proj[3,4]` to drop the old state.

Actually, now that I think of it, that's what the conditioning operator `//` is for. So I would define

```
def n_ticks(n, S):
    assert n >= 0, "Number of ticks must be non-negative."

    State = S
    for _ in range(n):
        State = clone(tick_light) // State

    return State
```


where I just successively update the state with a fresh clone of `tick_light`. Then I could do, say

```
pgd> n_ticks(10, frp(start_any_color))
An FRP with value <2, 5>
pgd> ten_ticks = _      # _ is the last value
```

and I could pass this FRP to `n_ticks` to continue.

```
pgd> n_ticks(30, ten_ticks)
An FRP with value <0, 4>
```

DANIELLE: That's great. Having the values of the FRPs is nice, but to make predictions, we would like to be able to find the Kinds of these FRPs. Does that work?

YOU: Let's see. To understand this, it will help me to start from a known state where we are just starting a  cycle.

```
pgd> start_green = constant(TrafficLight.GREEN, 0)
pgd> StartGreen = frp(start_green)
pgd> kind( n_ticks(0, StartGreen) )
<> ----- 1 ---- <0, 0>
pgd> kind( n_ticks(1, StartGreen) )
,---- 29/30 ---- <0, 1>
```

```

<> -|
    `---- 1/30 ----- <1, 0>
pgd> kind( n_ticks(2, StartGreen) )
    ,---- 0.93444 ----- <0, 2>
    |---- 0.032222 ----- <1, 0>
<> -|
    |---- 0.026667 ----- <1, 1>
    `---- 0.0066667 ----- <2, 0>
pgd> kind( n_ticks(3, StartGreen) )
    ,---- 0.00022222 ----- <0, 0>
    |---- 0.90330 ----- <0, 3>
    |---- 0.031148 ----- <1, 0>
<> -+---- 0.025778 ----- <1, 1>
    |---- 0.021333 ----- <1, 2>
    |---- 0.011778 ----- <2, 0>
    `---- 0.0064444 ----- <2, 1>
pgd> kind( n_ticks(100, StartGreen) )
...output omitted

```

As far as I can see, this is right. Each time, there is only a small probability of the light changing, and these accumulate so we get the chance of a yellow, then a red, then back to green. And on the second tick, we can see that the chance of a red is $1/30 \cdot 1/5$ with two changes in a row, as it should be. That last one has size 298, but glancing at the values, they make sense.

CARLOS: If you pay attention only to the colors what do you see.

YOU: I can transform those Kinds with the projection statistic `Proj[1]` to get the Kind of the color.

```




pgd> kind( n_ticks(10, StartGreen) ^ Proj[1] )
    ,---- 0.72831 ----- 0
<> -+---- 0.12186 ----- 1
    `---- 0.14983 ----- 2
pgd> kind( n_ticks(30, StartGreen) ^ Proj[1] )
    ,---- 0.51840 ----- 0





```

```

<> -+---- 0.091777 ---- 1
      `---- 0.38983 ----- 2
pgd> kind( n_ticks(50, StartGreen) ^ Proj[1] )
      ,---- 0.47336 ----- 0
<> -+---- 0.080086 ---- 1
      `---- 0.44655 ----- 2
pgd> kind( n_ticks(100, StartGreen) ^ Proj[1] )
      ,---- 0.46177 ----- 0
<> -+---- 0.076985 ---- 1
      `---- 0.46124 ----- 2
pgd> kind( n_ticks(500, StartGreen) ^ Proj[1] )
      ,---- 6/13 ---- 0
<> -+---- 1/13 ---- 1
      `---- 6/13 ---- 2

```

Interesting. We start at the beginning of a  cycle, so after a few ticks, we are much more likely to still be  because the probability of changing is so low. But as the number of ticks increases our predictions change, as though the system were “forgetting” that started out as . I can see with a little fiddling that above, say, 300 ticks the weights do not change to numerical precision.

If we wait long enough, we will predict  and  each 6/13 of the time and  1/13. We would expect  to be less likely as the system is more likely to switch out of yellow at any tick.

DANIELLE: So, you can make both short-term and long term predictions. If you were observing the traffic light from the street, you would see the color but not the full state. How would you compute your predictions in that case?

YOU: An interesting question. But before I answer that, something’s on my mind. I’ve been computing the Kinds of FRPs here, and it works. But I realize I could also compute the Kinds and then generate FRPs from them. I assume both ways would give the same result.

DANIELLE: They would. How would you compute the Kinds?

YOU: Well, instead of a conditional FRP, I would use a conditional Kind, and in fact, I already did that. Let me refactor my code a bit:

```

@conditional_kind
def tick_light_kind(state):
    color, ticks = state
    change_probability = change_on[color]

    return weighted_as(
        (color, ticks + 1),
        (next(color), 0),
        weights=[1 - change_probability, change_probability]
    )

@conditional_frp
def tick_light(state):
    return frp(tick_light_kind(state))

def n_ticks_kind(n, initial_state):
    assert n >= 0, "Number of ticks must be non-negative."

    state = initial_state
    for _ in range(n):
        state = tick_light_kind // state

    return state

def n_ticks(n, InitialState):
    return frp(n_ticks_kind(n, kind(InitialState)))

```


As before, the conditional Kind either adds to ticks or takes the next color with corresponding probabilities.

It's very similar to the FRP version, but it makes it easier to work with the Kinds to make predictions. And now `n_ticks` can take a Kind or an FRP in the second argument.

CARLOS: If `tick_light_kind` were hard to compute, then your original approach to `n_ticks` would be more efficient. But here, it's good.

YOU: OK, Danielle, back to your question. If I observed only the color of the traffic light, I would need to apply a conditional constraint to make my predictions.

DANIELLE: Right! Try it.

YOU: Using the more recent code, if I observed , I'd find the Kind of the state after, say, 30 ticks to be:

```
pgd> n_ticks_kind(30, start_any_color) | (Proj[1] == TrafficLight.GREEN)
,---- 0.047580 ---- <0, 0>
|---- 0.045730 ---- <0, 1>
|---- 0.043751 ---- <0, 2>
|---- 0.041604 ---- <0, 3>
|---- 0.039242 ---- <0, 4>
<> -+---- 0.036610 ---- <0, 5>
|---- 0.033640 ---- <0, 6>
|---- 0.030255 ---- <0, 7>
|---- 0.026360 ---- <0, 8>
|---- 0.021841 ---- <0, 9>
`---- 0.63339  ----- <0, 10>
```

CARLOS: Nice! Here's a challenge. Your conditional FRP makes the same decision no matter how long the light has been at its current color. Can you modify your work so that it uses ticks in a reasonable way?

YOU: This only requires changing tick_light_kind.

```
def stay_factor(ticks):
    return numeric_exp(-ticks / 2)  # must be >= 0

@conditional_kind
def tick_light_kind(state):  # only change is this factor //
    color, ticks = state      #                               vv
    change_probability = 1 - (1 - change_on[color]) * stay_factor(ticks)

    return weighted_as(
        (color, ticks + 1),
```



```

        (next(color), 0),
        weights=[1 - change_probability, change_probability]
    )

```

Now the change probability starts off as before when `ticks` is 0 but gets closer to 1 as `ticks` grows from. We can set the function `stay_factor` to any non-negative function.

CARLOS: That's works, very good. It can be a bit hard to understand what the choice of `stay_factor` means for how long the light will stay one color. An alternative is to choose a Kind for how long the light will stay as the current color when the color changes and to change the state to hold that information.

YOU: So the state would become $\langle c, n, \ell \rangle$ where c and n are like before and ℓ is the number of ticks remaining until the color changes.

I would need a Kind for each color representing the time that the light stays that color. I'll make that a dictionary `duration` that maps colors to Kinds.

That would give something like this:

```

@conditional_kind
def tick_light_kind(state):
    color, ticks, remaining = state

    if remaining > 0:
        return constant(color, ticks + 1, remaining - 1)

    return duration[color] ^ Fork(next(color), 0, Id)

```

The `Fork` makes a Kind with the next color and 0 in the first two components and the duration in the third. I would then have to modify the initial state Kind in the same way, but everything else should work as is.

CARLOS: Excellent. You can play with these ideas more with code in the `frplib.examples.traffic_light` module.



YOU: Thanks. That gives me a better idea of how these systems work. Danielle had mentioned, though, that you could make the lights respond to current conditions. How would that work?

DANIELLE: Systems have a *state* that describes their internal configuration, which is what you've been exploring here. In practice, we often also want them to respond to external *inputs* to produce *outputs* we can use.

YOU: What distinguishes inputs and outputs from state?

DANIELLE: The inputs can influence the state, and the outputs can be derived from the state. But in general, we think of input and output as separate parts of the system. Let me give an example.

Our traffic light is at an intersection in which cars are arriving, stopping, and passing through. Define two-dimensional FRPs

- $T^{(n)}$, representing the number of cars waiting in the direction of the light (component $T_1^{(n)}$) and in the other direction (component $T_2^{(n)}$), after n ticks of the clock.
- $M^{(n)}$, representing the number of cars who pass through the light in the direction that is . (For simplicity, assume cautious drivers who stop on .)

The $T^{(n)}$'s comprise a system, which we view as inputs to the light, and $M^{(n)}$'s comprise a system, which we view as outputs,

YOU: If the $T^{(n)}$'s are inputs, does the state just include the number of cars waiting in each direction.

CARLOS: It could, but to sharpen the ideas, assume that there are sensors on the traffic light that estimates the number of waiting cars but max out at ten cars.

Now our state looks like $\langle c, t, w_1, w_2 \rangle$ where w_1 and w_2 are the values recorded from the sensors in $[0 \dots 10]$.

DANIELLE: Let $L^{(n)}$ be the conditional FRP representing the traffic light after n ticks, which takes the number of waiting cars in each direction as input. Then we have a 6-dimensional FRP $S^{(n)} = T^{(n)} \triangleright L^{(n)}$.


The update of $T^{(n)}$ to $T^{(n+1)}$ depends on three things:

1. how many cars are waiting,
2. whether the green light stays on during tick $n + 1$, and
3. how many cars arrive during tick $n + 1$.

At the same time, we can design the traffic light to adjust its timing based on the estimated number of waiting cars that its sensors read. If w_1 is much bigger than w_2 ,

then the  will tend to remain on longer, and shorter in the reverse case.

YOU: I get it. The T and L systems are coupled; they depend on each other. So we can write a rule that transitions $S^{(n)}$ to $S^{(n+1)}$ along the lines you just described.

CARLOS: Right! And similarly, the output process $M^{(n+1)} = \psi(S^{(n)})$ for some statistic. (Note the $n + 1$ on the left and the n on the right.) The number of cars passing through depends on how many cars are waiting and which light, if any, is .

YOU: By taking appropriate mixtures and transforming with well-chosen statistics, we build the entire system. We just need to describe the Kinds of the traffic-light transitions, the number of cars that arrive in each direction, and the number of cars that get through during a given tick. And that breaks the problem into three separate things that are easier to understand and specify. Very nice!

DANIELLE: Exactly. There are some details, but you could code it up with what you already know. And the playground has some other tools to help, which I'm sure you'll encounter soon.

YOU: Fantastic. It occurs to me that we are making an assumption about these systems: that to find the next state we only need to know the current state but not the earlier history.

CARLOS: You're absolutely right about that. It is a huge assumption!

DANIELLE: It's possible to make systems whose evolution depends on their earlier history – even their whole history. Imagine if the next state is determined by a conditional Kind that uses not only the current state but the state before that and the state before that. Such systems can be useful, but the farther back in that history we have to go, the harder it is to do the analysis and computations.

CARLOS: For many practical problems, we can get what we need by making the assumption you mentioned.

DANIELLE: That assumption is called the **Markov property**. A system has the Markov property if its future evolution depends only on the current state but not on how it got there.

CARLOS: More formally, the Markov property means that if you know the current state, then your predictions about the system's future do not change if you also learn its full history.

We say that *given* the current state, the future and past are independent.

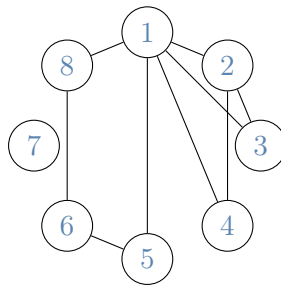
YOU: What’s an example, besides the traffic light, of a system with the Markov property?

DANIELLE: Are you familiar with graphs? The graphs with nodes and edges, I mean.

YOU: A little bit, yes.

CARLOS: A graph is a mathematical structure that describes pairwise relationships among several entities.⁶⁰ A graph has a set of nodes representing the entities and a set of edges representing the relationships between pairs of entities.

Here’s an example with eight nodes and various edges.



We call this graph *simple*, because there is at most one edge connecting any two nodes); *undirected* because the edges do not have a preferred direction; and *without loops* because there are no edges connecting a node to itself.

DANIELLE: Let’s build a system that describes a “random walk” on this graph. Our state will be the number of the current node, $\langle n \rangle$ for $n \in [1..8]$. If we are currently at node n , the Kind of the next state will be either `constant(n)` if node n has no edges connected to it or `uniform(neighbors(n))` where `neighbors(n)` lists all other nodes connected to n . We call each such update a **transition** or step, even if the state itself does not change.

How would you build a system like that?

YOU: I see that it’s not that different from what did with the traffic light. I’ll write everything in terms of the Kinds, and we can apply `frp` where needed to generate the FRPs.

The conditional Kind given that we are at a particular node n returns a Kind with equal weight on every node connected to the input node n by an edge.

⁶⁰The Random Graphs example in Section 2 shows various examples. See Interlude F Examples F.2.18 and F.2.19 for an overview and Interlude G for a detailed discussion.

For the random walk, we successively update the Kind of the current state (starting from the initial state) with the conditional Kind.

```
@conditional_kind
def next_node(node):
    adjacent = neighbors(node)
    if len(adjacent) == 0:
        return constant(node)
    return uniform(adjacent)

def random_walk(start, n_steps=1):
    assert n_steps >= 0, "Number of steps must be >= 0."

    current = start
    for _ in range(n_steps):
        current = next_node // current

    return current
```

CARLOS: Yes, the similarity with the traffic light is not a coincidence. Your line `current = next_node // current` is just an expression of the Markov property. The next node only depends on the current state, not the history of how you got there.

Notice also that your `random_walk` function works equally well when `start` is a Kind or an FRP.

YOU: I get that this lets us describe the system, but what can we do with this.

CARLOS: We can answer questions about the system like: How likely is the system to move from one particular state to another? How long before the system visits a specific state? How many times is the system in a particular state during the first n steps? And many more.




YOU: I understand how to answer a question like “Will the system be in node 8 after 100 steps if it starts at node 1?” Just run `random_walk(constant(1), 100)` and look at the weight on branch $\langle 8 \rangle$.

But I do not see how to answer those other questions.

DANIELLE: That question opens some very interesting doors. You should go talk to Erin and Fuyuan over in Building Y. They specialize in those techniques, and I think you'll enjoy a chat.

But in the meantime, let me illustrate a simple approach here. Consider two questions:

1. If we let the system run for a long time, does it settle down into an equilibrium that is independent of where it started, and if so, what state will it be in?
2. If the system starts at node 1, how long (if ever) until it reaches node 8?

For the first question, you will recall that the traffic light system you built did exactly that. After more than about 200 steps, the chance of being  or  or  did not change noticeably. Using your `random_walk` function, we can do:

```
pgd> random_walk(constant(1), 2500)
,---- 5/18 ---- 1
|---- 3/18 ---- 2
|---- 2/18 ---- 3
<> -+---- 2/18 ---- 4
|---- 2/18 ---- 5
|---- 2/18 ---- 6
`---- 2/18 ---- 8

pgd> random_walk(constant(3), 2500) # same for any node except 7
,---- 5/18 ---- 1
|---- 3/18 ---- 2
|---- 2/18 ---- 3
<> -+---- 2/18 ---- 4
|---- 2/18 ---- 5
|---- 2/18 ---- 6
`---- 2/18 ---- 8

pgd> random_walk(constant(7), 2500)
<> ----- 1 ---- 7
```

We can see that node 1 is a juncture between two parts of the graph, so we'd expect the system to spend more time there. And that's what we see. For any starting node except 7, the long-term behavior of the system is the same. Indeed, the numerator in those weights is just the number of neighbors of each node. For node 7, there's

nowhere to go, so we fully understand the behavior if the system starts there.

For question 2, Erin and Fuyuan will show you a beautiful answer, and you can see from the answer to question 1 that we *will* eventually visit node 8 unless we start in node 7.

Still, if we modify the state of our system, we can get an output that answers our question. Our new state will have the form $\langle n, v, t \rangle$ where n is the current node as before, v is 1 if we have ever *visited* node 8 or 0 otherwise, and t is the number of steps (the “time”) until we first visit node 8.

We can modify your code as follows:

```
@conditional_kind
def next_node(state):
    "Version of next_node that tracks whether we visited node 8."
    node, visit, time = state
    adjacent = neighbors(node)
    if len(adjacent) == 0:
        return constant(state)

    node_kind = uniform(adjacent)
    if visit == 1:
        return node_kind ^ Fork(Id, visit, time)
    if node == 8:
        return node_kind ^ Fork(Id, 1, time + 1)
    return uniform(adjacent) ^ Fork(Id, 0, time + 1)
```

All we do is keep track of whether we’ve visited 8 and either increment or freeze the time accordingly. Now, we can look at the Kind of the time component, using -1 to indicate that we have not visited node 8.

```
pgd> random_walk(constant(1, 0, 0), 360) ^ IfThenElse(Proj[2] == 1, Proj[3], -1)
... output of size 359 omitted
pgd> E(random_walk(constant(1, 0, 0), 360) ^ IfThenElse(Proj[2] == 1, Proj[3], -1))
23/2
```

The 360 here is arbitrary, long enough to be “long term” but short enough to be

computed quickly. We want this as large as possible, infinite really, but in practice it does not need to be very large. Above 360, for instance, there is at most negligible change. Because we know in this case that the system will eventually visit node 8, the weight on $\langle -1 \rangle$ tells us how good our approximation is. Here, it's about 4×10^{-15} , which is plenty close to zero. (Other techniques do not require even this accommodation.)

Applying the expectation operator E , we get our best prediction of how long until we visit node 8: 11.5 steps on average.⁶¹

⁶¹Section 7 will discuss interpretation of this number in detail.

YOU: That is very cool. So by augmenting the state, you can summarize many different aspects of the history and make predictions about them.

CARLOS: Right, we have a lot of flexibility to define the state in ways that let us answer interesting questions. And this graph example has broad application to lots of real problems.

YOU: I appreciate all your time here. I've got a lot to think about. I'll certainly wander over to Building U after the tour.

DANIELLE: You won't regret it. It is a bit hard to find, so wear comfortable shoes.

6.3 A Dialogue on Solutions and Simulation

After a long walk, you arrive in the basement of Building U. Walking through dank hallways with dripping pipes and flickering fluorescent lights, you discover a shiny, modern lab. Inside, you see Warehouse researchers Erin and Fuyuan, who look up at you quizzically.

ERIN: Do you have the pizza?

YOU: Pizza? No, I'm hoping to talk with you if you have a little time.

FUYUAN: That's another pizza delivery failure. We should put up signs.

YOU: Signs would help.

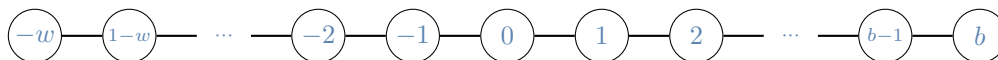
ERIN: Why are you here?

YOU: Carlos and Danielle recommended that I talk to you about techniques for answering questions about random systems.

FUYUAN: They showed you the random walk on graphs, didn't they?

YOU: Yes.

ERIN: (*moving toward a large display*) That's a good start. Think about this graph for some positive integers w and b :



FUYUAN: Imagine that a gambler comes to a casino with $\$w$ and places a series of identical, independent $\$1$ bets until either losing all her money or increasing her wealth by $\$b$. The state of the system – the node in the graph – is the *change* in the gambler's wealth. How likely is the gambler to be ruined – lose all her money – in this game?

YOU: If we had $w = b$ and if she had the same chance of winning and losing each bet, then she would be equally likely to be ruined and achieve her goal by the symmetry between winning and losing. As b grows (shrinks) relative to w , her chance of achieving her goal should get smaller (larger), I'd guess.

ERIN: Good. We'll do that case first, but we want to handle cases where the bet's outcomes are not evenly weighted, like real casino games for instance.

FUYUAN: We can consider three broad approaches to this problem: (i) solve for an exact solution, (ii) solve for an approximate solution and make the approximation error small if possible, and (iii) simulate the system to estimate the solution. Let's start with (ii) and (iii) to set things up.

YOU: From my discussion with Danielle and Carlos, I think I know how to do that.

```
pgd> def gamble_with(wealth, goal):
...>     assert wealth >= 0 and goal >= 0
...>     bet = either(-1, 1)
...>
...>     @conditional_kind(codim=1, target_dim=1)
...>     def wealth_change(state):
...>         if state == -wealth or state == goal:
...>             return constant(state)
...>         return bet ^ (__ + state)
...>
...>     return wealth_change
```

```

pgd> def gamblers_walk(next_state, start=constant(0), n_steps=1):
...>     current = start
...>     for _ in range(n_steps):
...>         current = next_state // current
...>     return current

```

Here, I wrapped the conditional Kind in a factory function so that we could set the gambler's initial wealth and goal. If the system gets to state $-w$ or b , it stays there; otherwise, it goes up or down according to the output of the bet. The `gamblers_walk` function is like what I did earlier with the traffic lights, except I take the conditional Kind as an argument and by default start at state 0.

Then, I think I can do both (ii) and (iii):

```

pgd> K = gamblers_walk(gamble_with(10, 15), n_steps=3000)
pgd> C = frp(K) ^ Cases({-10: -1, 15: 1}, 0)
pgd> FRP.sample(10_000, C)
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 6043 | 60.43% |
| 1      | 3957 | 39.57% |
+-----+-----+-----+
pgd> K ^ Cases({-10: -1, 15: 1}, 0)
      ,---- 3/5 ---- -1
<> -+---- 0/5 ---- 0
      `---- 2/5 ---- 1

```

The `Cases` built-in statistic converts the original values to -1 for ruin, 1 for success, and 0 for unresolved. And we can see that after 3000 steps, there is a negligible chance that the Gambler has not achieved her goal or ruin.

ERIN: Both strategies (ii) and (iii) work well in this case as you say, and we can even guess at the *exact* solution here. When $w = b$, we get $1/2$ each for goal and ruin; when $w = 10$ and $b = 15$, we get $2/5$ and $3/5$. Can you guess?

YOU: I think I have it, but let me try a few experiments

```

pgd> def ruin_or_goal(w, b, steps=3000):
...>     s = Cases({-w: -1, b: 1}, 0)
...>     return s(gamblers_walk(gamble_with(w, b), n_steps=steps))
pgd> ruin_or_goal(5, 15)
      ,---- 3/4 ---- -1
<> -+----- 0/4 ---- 0
      `---- 1/4 ---- 1
pgd> ruin_or_goal(5, 20)
      ,---- 4/5 ---- -1
<> -+----- 0/4 ---- 0
      `---- 1/5 ---- 1
pgd> ruin_or_goal(19, 1)
      ,---- 0.050000 ----- -1
<> -+----- 1.4481E-17 ---- 0
      `---- 0.95000 ----- 1

```

My guess is that the probabilities of ruin and goal are $\frac{b}{w+b}$ and $\frac{w}{w+b}$.

FUYUAN: That's right.

YOU: Shouldn't this approach always work.

FUYUAN: In a sense it does work, and these are very useful techniques. However in many cases, it requires many more runs to get a good approximation, and it can be harder to guess the exact answer.

Two examples are worth considering: bets where winning and losing do not have equal weight and trying to predict *how many bets* it takes before the gambler is ruined or achieves her goal.

YOU: The first is a simple change to `bet` and add a parameter:

```

pgd> def gamble_with(wealth, goal, win_to_lose=1): # <<- added parameter
...>     assert wealth >= 0 and goal >= 0
...>     bet = either(1, -1, win_to_lose) # <<- only change from earlier
...>
...>     @conditional_kind(codim=1)
...>     def wealth_change(state):
...>         if state == -wealth or state == goal:

```

```

...>         return constant(state)
...>         return bet ^ (__ + state)
...>
...>         return wealth_change

```

Then I'll add the win-to-lose ratio to `ruin_or_goal`, another simple change:

```

pgd> def ruin_or_goal(w, b, win_to_lose=1, steps=3000):
...>     s = Cases({-w: -1, b: 1}, 0)
...>     return s(gamblers_walk(gamble_with(w, b, win_to_lose), n_steps=steps))

```

And then run the simulation, for example with a win-to-lose weight ratio of 16/17:

```

pgd> ruin_or_goal(5, 20, '16/17')
,---- 0.90032 ----- -1
<> -+---- 7.3879E-12 ---- 0
    ^---- 0.099679 ----- 1

```

The approximation must be pretty good because the weight on 0 is so small, but I see what you mean though. Not easy to guess what those numbers mean.

ERIN: Nice. And for the second example?

YOU: We have to change the state to keep track of how many steps we have taken but stop counting when we hit either ruin or goal.

Is there a way to avoid rewriting these functions every time we change state?

ERIN: For the first, you are right; try it. For the second, yes to some degree, but it requires a little fancier programming. It's probably best if we don't go down that rabbit hole right now, but an example:

```

def markov_transition(start, next_state):
    def do_steps(n_steps=1):
        current = start
        for _ in range(n_steps):
            current = next_state // current
        return current
    return do_steps

```

Now, calling `markov_transition` with the Kind of the initial state and the conditional Kind of the transition will return a *function* that computes the Kind of the state after any number of steps.

YOU: Thanks, that's encouraging. I'm guessing that works for any system with the Markov property.

ERIN: Yes

YOU: OK, here's a version that tracks the time until ruin or goal.

```
pgd> def gamble_time(wealth, goal, win_to_lose=1): # <<- added parameter
...>     assert wealth >= 0 and goal >= 0
...>     bet = either(1, -1, win_to_lose) # <<- only change from earlier
...>
...>     @conditional_kind
...>     def wealth_change(state):
...>         delta_wealth, time = state
...>         if delta_wealth == -wealth or delta_wealth == goal:
...>             return constant(state)
...>         return bet ^ (__ + delta_wealth) ^ Fork(Id, time + 1)
...>
...>     return wealth_change

pgd> go = markov_transition(constant(0,0), gamble_time(10, 15))
pgd> t3000 = Proj[2](go(3000))
```

Oh, that took a little while.

ERIN: There are many paths, so there is a small chance of it taking a while. It can help to use `clean` to eliminate the negligible branches or use `E` to get a simpler prediction.

```
pgd> E(t3000)
149.9999999725631
```

which we can guess should be 150 if exact. This is our best prediction about how long it will take the gambler to reach ruin or goal.

FUYUAN: All this is prologue to looking at strategy (i). It's good to see that we can still compute good answers without an exact solution but it takes a little work and judgment.

To find an exact solution in this problem, we are going to solve *several similar problems simultaneously*. This will give us an equation that we can solve exactly.

ERIN: It will help to see it in action. We'll start with the gambler's ruin problem and a couple other concrete cases, but the equation we derive will work for a huge variety of problems, as we will see.

FUYUAN: We will write $\text{GamblersRuin}\langle w, b, r \rangle$ to denote the gambler's ruin problem with initial wealth w , goal b , and win-to-lose ratio r .

Our goal is to predict the number of bets it takes before the gambler gets to ruin or her goal. At the start, her net change in wealth is 0, i.e., she starts in state 0. But to find this prediction starting at 0, we will consider the versions of the same problem *for every starting node*.

Let T_s be an FRP representing the number of bets ("time") it takes before the gambler gets to ruin or goal when her starting state is $s \in [-w..b]$. We loosely call the number of bets "time" because it's easier and makes sense. Our best prediction of T_s 's value is $\mathbb{E}(T_s)$.

ERIN: Define a function f on $[-w..b]$ by

$$f(s) = \mathbb{E}(T_s). \quad (6.1)$$

We really want to find $f(0)$, but to do that we will solve for the *whole function*.

What do we know about the function f ?

YOU: I suppose that if we start at ruin or goal, we don't have to place any more bets. So, $f(-w) = 0 = f(b)$.

FUYUAN: Good, yes. Now we come to the key idea: what does the problem look like *after one step*.

YOU: What does the problem look like??

FUYUAN: If $-w < s < b$, what states can we reach after one step?

YOU: Either $s - 1$ or $s + 1$ with weights 1 and r .

ERIN: And from either of those states, what is the expected time until goal or ruin.

Remember the Markov property.

YOU: Hmm. Once we are in state $s - 1$, our expected time is $f(s - 1)$ and once we are in $s + 1$, our expected time is $f(s + 1)$.

FUYUAN: And it took one step to get there. So what is the *Kind* of the expected time when you start in state s ?

YOU: (*excited*) Wait. We start in state s , and the state we move to is represented by an FRP, call it X_1 . The function $1 + f$ is solving our problem but it's just a function, so we can use it as a *statistic* and compute the Kind of $1 + f(X_1)$. Damn.

This gives us the Kind, in canonical form:

$$\langle \rangle - \left[\begin{array}{l} \frac{1}{1+r} \text{ --- } \langle 1 + f(s - 1) \rangle \\ \frac{r}{1+r} \text{ --- } \langle 1 + f(s + 1) \rangle \end{array} \right] \quad (6.2)$$

ERIN: And this has expectation⁶²

$$(1 + f(s - 1)) \frac{1}{1+r} + (1 + f(s + 1)) \frac{r}{1+r} = 1 + \frac{f(s - 1) + rf(s + 1)}{1+r}.$$

⁶²A general formula is given Section 7, but take this on faith for now.

FUYUAN: This is our best prediction of the time it takes to reach ruin or goal, but that is also the definition of $f(s)$. So ***the two must be equal!*** That is, for every $s \in (-w \dots b)$ we have⁶³

$$f(s) = 1 + \frac{f(s - 1) + rf(s + 1)}{1+r} \quad (6.3)$$

⁶³ $(a \dots b)$ is the set of integers from a to b excluding a and b . See Section F.1.2.

with $f(-w) = 0 = f(b)$.

This gives us $w + b - 1$ equations in $w + b - 1$ unknowns, so we can solve for f and then just read off $f(0)$, the answer to our original problem.

YOU: Whoa.

ERIN and FUYUAN: (*nods*)

ERIN: For instance, if $r = 1$ (winning and losing equally likely), this can be re-written

as the system of equations:

$$f(s+1) - 2f(s) + f(s-1) = -2 \quad \text{for } s \in (-w \dots b) \quad (6.4)$$

$$f(b) = 0 \quad (6.5)$$

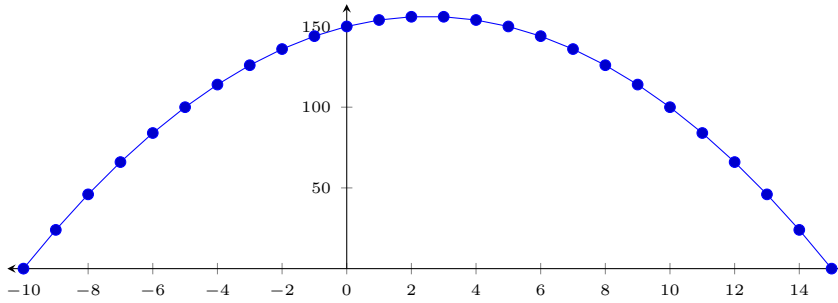
$$f(-w) = 0. \quad (6.6)$$

The expression on the left side of the first equation is called the “second difference” of f , $\Delta^2 f$, at $s-1$. Differences are a discrete analogue of derivatives,⁶⁴ so we have a “second-order difference equation” for f with values at the boundary specified. We can solve this difference equation to find

⁶⁴The discrete calculus is developed in the “Sequences and Streams” examples and puzzles in Section F.2.

$$f(s) = (b-s)(w+s). \quad (6.7)$$

So, in general $f(0) = bw$, and when $b = 15$ and $w = 10$, f looks like



with $f(0) = 150$ as you found earlier.

YOU: That is very cool. I do wonder how broadly applicable this approach is. For instance, earlier we guessed the probability that the gambler is ruined. Could we find it exactly?

FUYUAN: As with anything, there are trade-offs between what we must assume and the tractability of the problem. But the approach is quite general.

Suppose we have a system with the Markov property that visits a finite set of states. Whenever the system is in state s , the Kind of the next state is K_s , and these Kinds can vary from state to state.

Now imagine there are two sets of states \mathcal{S}_0 and \mathcal{S}_1 , and we want to know the chance that the system visits a state in \mathcal{S}_1 before any state in \mathcal{S}_0 .

YOU: So in the $\text{GamblersRuin}\langle w, b, r \rangle$ problem, we can take $\mathcal{S}_1 = \{-w\}$ and $\mathcal{S}_0 = \{b\}$ and the Kind K_s is

$$\langle \rangle \begin{cases} \xrightarrow{\frac{1}{1+r}} \langle s-1 \rangle \\ \xrightarrow{\frac{r}{1+r}} \langle s+1 \rangle \end{cases}$$

FUYUAN: Exactly. And similarly to what we did earlier, define $f(s)$ to be the probability of visiting \mathcal{S}_1 before \mathcal{S}_0 *when we start the system in state s* . Then, using exactly the same logic as earlier, we have

$$\begin{aligned} f(s) &= 0 && \text{if } s \in \mathcal{S}_0 \\ f(s) &= 1 && \text{if } s \in \mathcal{S}_1 \\ f(s) &= \mathbb{E}(f(K_s)) && \text{if } s \notin \mathcal{S}_0 \cup \mathcal{S}_1, \end{aligned} \tag{6.8}$$

where $\mathbb{E}(f(K_s))$ is our best prediction of the value of an FRP with Kind $f(K_s)$.

YOU: But we don't know f .

ERIN: True, but we can write the transformed Kind $f(K_s)$ in terms of f 's value, just like you did in (6.2). And we get one equation and one unknown for each state not in $\mathcal{S}_0 \cup \mathcal{S}_1$.

YOU: So if we try this with $\text{GamblersRuin}\langle w, b, 1 \rangle$ we get

$$\begin{aligned} f(b) &= 0 \\ f(-w) &= 1 \\ f(s) &= \frac{1}{2}f(s-1) + \frac{1}{2}f(s+1) \quad \text{otherwise,} \end{aligned}$$

so

$$f(s) - f(s-1) = f(s+1) - f(s).$$

The differences of f are constant, meaning that f must shrink *linearly* from value 1 at $-w$ to value 0 at b . I think this means

$$f(s) = \frac{b-s}{b+w} \tag{6.9}$$

giving

$$f(0) = \frac{b}{b+w} \quad (6.10)$$

as we guessed earlier.

ERIN: Good. Now, imagine you are in a room that looks like the one in Figure 43. You start on a tile in the middle of the room (marked green) and move North, South, East, or West randomly, with equal weights. The red tiles are lava – you don't want to touch those – and the blue tiles are cool water through which you can swim to a pleasant beach resort. Will you end up sipping Mai Tais on the beach or doing a Gollum lava-dive?

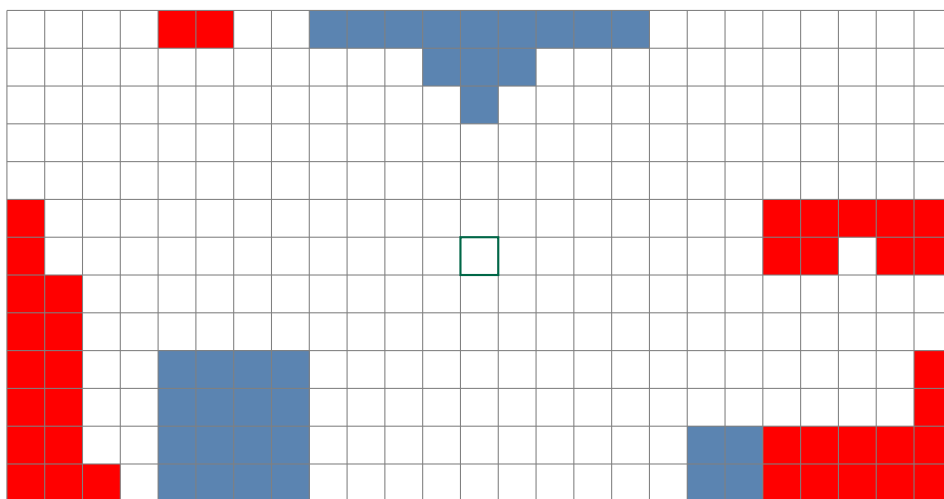


FIGURE 43. A room with lava (ouch) and a cool swim to the beach (yay).

YOU: So \mathcal{S}_0 contains all the lava tiles and \mathcal{S}_1 the water tiles. Put the starting tile at $\langle 0, 0 \rangle$. If we are at tile $s = \langle x, y \rangle$ away from a wall or corner, the Kind K_s is

$$\langle \rangle \begin{cases} \frac{1}{4} & \langle x-1, y \rangle \\ \frac{1}{4} & \langle x, y-1 \rangle \\ \frac{1}{4} & \langle x, y+1 \rangle \\ \frac{1}{4} & \langle x+1, y \rangle \end{cases}$$

So for such tiles not in lava or water, (6.8) becomes

$$f(x, y) = \frac{f(x-1, y) + f(x, y-1) + f(x, y+1) + f(x+1, y)}{4},$$

At a wall, only three neighbors are reachable, and at a corner only two. The weights in these cases are $1/3$ and $1/2$ respectively. Over all tiles, we have same number of unknowns as equations, so we can in principle solve for f .

FUYUAN: We can. In fact, we've put some of our methods for this in a playground module. You give it a conditional Kind (built from the K_s 's), the sets on which you know the answer, and the known answers on those sets. So to find the probability of hitting water before lava:

```
pgd> from frplib.examples.dirichlet import *
pgd> f = solve_dirichlet_sparse(K_NSEW, states=lava_room.states,
                               fixed=lava_room.fixed, fixed_values=(0, 1))
pgd> f(0, 0)          # starting at (0,0)
0.836882
```

Use $f(s) = 1 + \mathbb{E}(f(K_s))$ to get the expected time to hit either lava or water:

```
pgd> end_tiles = lava_room.fixed[0].union(lava_room.fixed[1])
pgd> f_time = solve_dirichlet_sparse(K_NSEW, alpha=1, states=lava_room.states,
                                     fixed=[end_tiles], fixed_values=[0])
pgd> f_time(0, 0)  # starting at (0, 0)
65.5347
```

ERIN: Here's a puzzle for you to try later. (We have other things to talk about now.)

Puzzle 46. Theseus is trapped in a labyrinth, as is his wont, and a fierce creature is trapped with him. Both start at distinct locations, and each moves randomly from their current juncture with equal weight on all available directions.

If Theseus and the creature ever meet, Theseus will be eaten. But Theseus (not the creature) can escape through a small opening at one specific juncture.

Create a small labyrinth and compute the probability that Theseus escapes.

YOU: Create a labyrinth?

ERIN: A labyrinth is just a graph. Each juncture is a node, and each path from a juncture to its neighbor is an edge.

In fact, the graphs we are working here are actually *directed* graphs with loops. The Kinds K_s can be different for each state, so it is possible that we can get from state s to s' in one step but not s' to s . It is also possible to stay in the same state, which is an edge from s to s . Each branch of the Kind K_s is associated with one (directed) edge that leaves node s in the graph.

We can actually solve a somewhat more general system than Fuyuan showed in (6.8). We have a set of states \mathcal{S} (nodes in our graph) that is partitioned into disjoint sets $\mathcal{S}_0, \mathcal{S}_1, \dots, \mathcal{S}_m$ for some m . We have a function f on \mathcal{S} whose value on each \mathcal{S}_i is known to be c_i for $i < m$ and that satisfies

$$\begin{aligned} f(s) &= c_i && \text{if } s \in \mathcal{S}_i \text{ for } i \in [0..m) \\ f(s) &= \alpha + \beta \mathbb{E}(f(K_s)) && \text{if } s \in \mathcal{S}_m, \end{aligned} \tag{6.11}$$

for fixed numbers α, β .

Setting $m = 2$, $c_0 = 0$, $c_1 = 1$, $\alpha = 0$, and $\beta = 1$, gives our prediction of whether the system hits \mathcal{S}_1 before \mathcal{S}_0 . Setting $m = 1$, $c_0 = 0$, $\alpha = 1$, and $\beta = 1$, we get our predicted time until the system hits \mathcal{S}_0 . And so on. Our `solve_dirichlet` will handle all those problems.

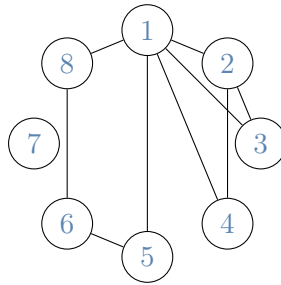
YOU: That's useful to know, thanks. I do have two questions. First, this lets us compute specific predictions (expectations), but can I use this to find the Kind itself. Second, Carlos and Danielle showed me a case where the system seemed to come to an equilibrium, but there's no "hitting" any set of states. Can we handle that?

FUYUAN: Good questions. The general answer to both is yes, but to keep things concise-ish, let's tackle an example of both at once.

Notice that if the system can get *stuck* in two distinct states, like the gambler's ruin, then it can't really get to an equilibrium. It can get stuck in one or the other, we're not sure which. But we've solved problems like that, so let's consider cases where there is a long-run equilibrium.

What was the graph that Carlos and Danielle showed you?

YOU: It was this



FUYUAN: OK, that's a good candidate, but let's ignore node 7. You can't reach it from the others and if you start there, you stay there, which is an equilibrium but a trivial one.

ERIN: Like the Hotel California

FUYUAN: *(smiles)* Imagine we have a system that describes a random walk on this graph. What does it mean for the system to be in “equilibrium”?

YOU: It means that at some level things don't change. Of course, they *change* because if you are moving around the graph, but . . .

ERIN: That's the idea. Think of it this way: when you look at the system at some arbitrary time, the current node is random, and it has some Kind K . Equilibrium means that that Kind doesn't change. The system might move among the nodes, but if the system is in equilibrium, your *prediction* about what node it will be after 100, 1000, 10000 steps will be *the same*.

YOU: That makes sense.

FUYUAN: So we do the same logic that led to our equations (6.11): we think about what happens in a *single step*.

Let S be the conditional Kind of the next node given the current node. If an FRP representing the current node has Kind K , then an FRP representing the next node has Kind $S \mathbin{\mathbb{I}} K$, and the assumption that the system is in equilibrium means that

$$K = S \mathbin{\mathbb{I}} K. \quad (6.12)$$

We just solve this equation for K .

We say that K is a **fixed point** of the function $\langle k \rangle \mapsto S \mathbin{\mathbb{I}} k$, because if we evaluate

this function at K , it returns K right back. The function “keeps K fixed.”

ERIN: We can usually find this K by iterating. Start with an arbitrary Kind, like `constant(0)`, and do the following

```
pgd> k0 = constant(0)
pgd> k0 = S // k0
pgd> k0 = S // k0
pgd> ...
```

or just do it in a loop. This `k0` should convert to K .

But the graph has finitely many nodes, so we can solve for K ’s weights directly.

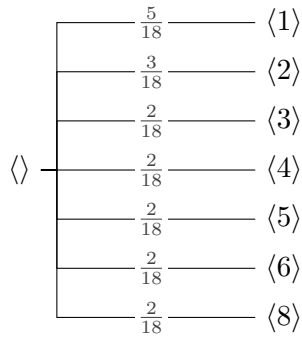
YOU: Solve?

FUYUAN: Sure. We know the values of K ; what we don’t know are the weights. Write out equation (6.12) in terms of those weights.

YOU: I see. So, to find the weight on node 1, say, in $S//K$, we need to add up the contribution from all nodes leading to 1.

$$\begin{aligned} w_1 &= \frac{1}{3}w_2 + \frac{1}{2}w_3 + \frac{1}{2}w_4 + \frac{1}{2}w_5 + \frac{1}{2}w_8 \\ w_2 &= \frac{1}{5}w_1 + \frac{1}{2}w_3 + \frac{1}{2}w_4 \\ w_3 &= \frac{1}{5}w_1 + \frac{1}{2}w_2 \\ w_4 &= \frac{1}{5}w_1 + \frac{1}{2}w_2 \\ w_5 &= \frac{1}{5}w_1 + \frac{1}{2}w_6 \\ w_6 &= \frac{1}{2}w_5 + \frac{1}{2}w_8 \\ w_8 &= \frac{1}{5}w_1 + \frac{1}{2}w_6. \end{aligned}$$

For instance, to get to 1, we must choose one neighbor out of three from node 2, and one out of two from each of nodes 3, 4, 5, and 8. We can solve these equations, yielding



as we saw before.

Let me do this in the playground to check my ideas. The Kind factory `prenormalized` is like `weighted_as` but assumes that weights with symbolic terms add up to 1.

```
pgd> S = conditional_kind({
    1: uniform(2, 3, 4, 5, 8),
    2: uniform(1, 3, 4),
    3: uniform(1, 2),
    4: uniform(1, 2),
    5: uniform(1, 6),
    6: uniform(5, 8),
    8: uniform(1, 6)
...> })
pgd> w = symbols('w1 w2 w3 w4 w5 w6 w8')
pgd> K = prenormalized(1, 2, ..., 6, 8, weights=w)
pgd> S // K
...output omitted
```

Yes, the same equations. Good.

ERIN: Great work!

YOU: If you cannot find an exact solution and have to resort to your strategies (ii) and (iii) that we discussed earlier, do you have ways to make those more efficient.

ERIN: There is not one general technique, but there are ways to speed things up a lot. The Markov property comes in handy. Here's a nice example.

We have an encrypted English text consisting of n characters $c_1 c_2 \cdots c_n$ that we would

like to decrypt. Assume two things for simplicity: (i) the text consists only of the 26 letters A-Z and spaces, and (ii) the encryption is with a *substitution cipher* that permutes the 27 characters and substitutes the true characters with the character permuted to that position. For example, if ABC is permuted as CAB, then every A in the true text will appear as C in the cipher text, every B as A, and every C as B. Our decryption d is thus be a permutation of the same 27 characters; we want the permutation that *inverts* the cipher. So, $d(A) = B$, $d(B) = C$, $d(C) = A$ would invert CAB, replacing every A in the cipher text with a B, every B with a C, and every C with an A.

YOU: The problem is that there are $27! = 10888869450418352160768000000$ possible d 's. That's a lot. Searching them all is practically impossible. Even searching enough to have a chance of finding the right one would take far too long.

FUYUAN: You're right, and that's why we need a bit of cleverness in our simulation. We've computed the frequency of character pairs $b(c, c')$ in a reference corpus⁶⁵ of English text, where we use special characters **start** and **end** to represent the beginning or end of the text.

⁶⁵See comments in the code for citations.

Using those frequencies, we give each d a *score*:

$$L(d) = b(\text{start}, d(c_1)) \prod_{i=2}^n b(d(c_{i-1}), d(c_i)) b(d(c_n), \text{end}).$$

We want to find the d that (at least approximately) *maximizes* the score L .

YOU: That helps you pick good or bad d 's, but don't you still have to search untold numbers of orderings of the characters?

ERIN: We create a system with the Markov property much like your random walk on a graph. Like the random walk, which settled into an equilibrium Kind with weights proportional to the number of neighbors a node has, this system will settle into an equilibrium proportional to the score $L(d)$.

As the system runs, we keep track of the state (d) with the largest score. As the system approaches equilibrium, that gets closer to the maximizer.

YOU: Can you be more specific about how you set up this system?

FUYUAN: The algorithm is simple. First, we choose a starting state, such as the d that is the identity permutation. Initialize the maximum score to $s = L(d)$ and best

decryption to $m = d$.

Then we repeat the following steps for as many iterations as we can:

1. If the current state is d with score $L(d)$, generate an FRP that randomly swaps two positions in d (e.g., ABC to CBA). Call the resulting permutation d' .
2. Compute $L(d')$ and let $p = \min(1, L(d')/L(d))$.
3. Generate an *event*⁶⁶ with Kind

$$\langle \rangle \begin{cases} \xrightarrow{1-p} \langle 0 \rangle \\ \xrightarrow{p} \langle 1 \rangle \end{cases}$$

⁶⁶Recall that an event is just an FRP with possible values 0 or 1.

4. If that event occurs, the system moves to state d' ; otherwise it stays in state d .
5. Set s to $\max(s, L(d'))$. If s increases, set m to d' .

YOU: So boiled down to the details, we randomly swap letters. If that increases the score, we move to that state; if not, we *might* move to that state, with a move more likely the bigger its score.

ERIN: That's all there is to it. Do you want to see it in action?

```
pgd> from frplib.examples.markov_decrypt import markov_decrypt, cipher1, clear1
pgd> cipher1
QXYVE WMAVDOCBJVLCKVP TGYFVCHYOVQXYVRUSNVFCI
pgd> decrypted, *_ = markov_decrypt(cipher1, iter=10_000)
pgd> decrypted
THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG
pgd> decrypted == clear1
True
```

You can look at the code, too. It's pretty simple.

YOU: That was surprisingly fast, and it worked.

ERIN: This technique works for a range of ciphers and it generalizes to many different kinds of problems.

YOU: Amazing. I really appreciate your time. The possibilities our exciting.

PIZZA PERSON: (*knocks on lab door*) Did someone order a pizza?

FUYUAN: Finally!

Checkpoints

After reading this section you should be able to:

- Identify situations where FRP sizes grow quickly.
- Explain the useful property of a “monoidal statistic”.
- Show how to compute a transformed Kind with a monoidal statistic quickly even when the original Kind has very large size.
- Use conditional Kinds/FRPs and mixtures to describe steps in a process.
- For some large Kinds, find an efficient way to answer targeted questions.
- Explain what the *state* of a system means.
- Define the state for some simple systems and use mixtures and statistics to update the state as the system evolves.
- Explain in words what the Markov property means.
- Simulate a system with the Markov property and if possible, compute the Kind of the state after some number of steps.
- Find simulated and approximate predictions for the time to hit some set of states or whether one set of states will be hit before the other.
- Explain how to construct an equation for an exact solution to such problems.
- Explain how to find the Kind of the state for a system in equilibrium.
- Describe the idea behind the Markov decryption simulation.

7 Predicting with Expectations

Key Take Aways

How much is an FRP worth?

To begin to answer that question, we consider how well we can predict the value of a scalar FRP. Following a long tradition, we express our prediction through a *price*. A larger predicted payoff corresponds to a higher price and a smaller (even negative) predicted payoff to a lower (even negative) price.

The FRP market lets us purchase *any number* of FRPs of the same Kind at a fixed price per unit. We can borrow and use unlimited funds with no interest but must pay back that loan when our FRPs' values are revealed.

If we can purchase a large number of FRPs of the same Kind at a price c that *essentially guarantees* us a profit, we call c an **arbitrage price** for those FRPs. If we have an opportunity to purchase FRPs at an arbitrage price, we would always take it – at scale.

The set of arbitrage prices for an FRP contains every number from $-\infty$ up to **but not including** a value r , which may be a real number or ∞ . This value r is the **risk-neutral price** for a scalar FRP. It is the smallest value that is bigger than all arbitrage prices for that FRP.

No reasonable person would offer us an arbitrage price to purchase FRPs because it would (essentially) guarantee them a loss. Nor would you accept an offer to pay *more* than the risk-neutral price, for it would (essentially) guarantee you a loss. But at the risk-neutral price, there are no guarantees; you may win or lose, and neither buyer nor seller has the advantage.

The term risk-neutral here means that the price is not sensitive to the risk of loss that you face or the degree of uncertainty in the FRPs value. When you can purchase as many FRPs as you like with interest-free funding, you are not sensitive to risk as we would be in real life. The risk-neutral price reflects a prediction of the value produced by an FRP, it is the best prediction in some sense and can be seen as a “typical value.”

For any FRP X , we define the **expectation** of X , denoted $\mathbb{E}(X)$. If X is a scalar (1-dimensional) FRP, its expectation is just its risk-neutral price. If X has dimension $n > 1$ and FRPs $\langle X_1, X_2, \dots, X_n \rangle$ are its scalar components, then $\mathbb{E}(X)$ is an n -tuple of numbers defined by the risk-neutral prices of its components:

$$\mathbb{E}(X) = \langle \mathbb{E}(X_1), \mathbb{E}(X_2), \dots, \mathbb{E}(X_n) \rangle, \quad (7.1)$$

$\mathbb{E}(X)$ depends only on $\text{kind}(X)$ not on X 's produced value.

The logic of risk-neutral prices gives us several key properties of expectations for any FRP X :

- **Constancy.** If X is a constant FRP, with one possible value v , then

$$\mathbb{E}(X) = v. \quad (7.2)$$

- **Scaling.** For any real number s ,

$$\mathbb{E}(sX) = s \mathbb{E}(X). \quad (7.3)$$

- **Ordering.** If all possible values of X are $\geq a$ and $\leq b$, then

$$a \leq \mathbb{E}(X) \leq b \quad (7.4)$$

- **Additivity.** If X_1, X_2, \dots, X_n are FRPs of *common dimension*, then

$$\mathbb{E}(X_1 + X_2 + \dots + X_n) = \mathbb{E}(X_1) + \mathbb{E}(X_2) + \dots + \mathbb{E}(X_n). \quad (7.5)$$

- **Substitution.** If X is an FRP and ψ and $\langle x \rangle \mapsto \zeta(x, \psi(x))$ are compatible statistics

$$\mathbb{E}(\zeta(X, \psi(X)) \mid \psi(X) = a) = \mathbb{E}(\zeta(X, a) \mid \psi(X) = a). \quad (7.6)$$

For any Kind K in canonical form with values v_1, \dots, v_m and corresponding weights p_1, \dots, p_m , then for any FRP X with Kind K :

$$\mathbb{E}(X) = p_1 v_1 + \dots + p_m v_m. \quad (7.7)$$

The expectation of a scalar FRP Y is the number that minimizes the *predicted squared prediction error*

$$\langle c \rangle \mapsto \mathbb{E}(|Y - c|^2).$$

The minimum value at $c = \mathbb{E}(Y)$ is called the **variance** of Y , denoted $\text{Var}(Y)$.

Two FRPs X and Y have the same Kind if and only if they have the same set of possible values and

$$\mathbb{E}(\psi(X)) = \mathbb{E}(\psi(Y)) \quad (7.8)$$

for *every compatible statistic* ψ . We can often find smaller collections of statistics that *determine* the Kind of FRPs.

A **probability** is the expectation of an event. It is a number in $[0_1]$ that measures our prediction of whether the event will occur. If V is an event, $\mathbb{E}(V)$ is called the *probability of V* . Events with higher probability are said to be more *likely* than events with lower probability.

The more we know about a process, the better we are able to predict its outcome. Indeed, we can think of *uncertainty* as quantifying the difficulty of making predictions. Uncertainty reflects limits to the accuracy with which we can predict an outcome. Sometimes these limits arise from our lack of information and sometimes they are intrinsic features of the system we are studying.

At one extreme, an outcome may be certain, and our prediction is perfect. For example, the constant FRP with Kind $\langle \rangle$ ——— $\langle 100 \rangle$ has only possible value (100 in this case), so we can predict with complete certainty that 100 is the value it will display when we push its button. Close to that is what we can call **essential certainty**. It is *possible* that all the air molecules in the room where you are sitting will spontaneously organize themselves in the corner of the room, leaving you in an effective vacuum, but for that to happen would require so many miraculous bounces that there is no reasonable need to factor that possibility into your day.

Toward the other extreme, an outcome may be uncertain with nothing to distinguish the possibilities. For instance, FRPs with Kind

$$\langle \rangle \begin{cases} \text{---} 1 \text{---} \langle 0 \rangle \\ \text{---} 1 \text{---} \langle 1 \rangle \end{cases}$$

can produce values 0 or 1, and in any sample of such FRPs there is no reason to expect one more than the other. Our uncertainty increases with the number and spread of the possible values. For instance, FRPs with Kinds

$$\begin{array}{cc} \begin{array}{c} \text{---} 1 \text{---} \langle -3 \rangle \\ \text{---} 1 \text{---} \langle -2 \rangle \\ \text{---} 1 \text{---} \langle -1 \rangle \\ \langle \rangle \text{---} 1 \text{---} \langle 0 \rangle \\ \text{---} 1 \text{---} \langle 1 \rangle \\ \text{---} 1 \text{---} \langle 2 \rangle \\ \text{---} 1 \text{---} \langle 3 \rangle \end{array} & \begin{array}{c} \text{---} 1 \text{---} \langle -10000 \rangle \\ \text{---} 1 \text{---} \langle -100 \rangle \\ \text{---} 1 \text{---} \langle -1 \rangle \\ \langle \rangle \text{---} 1 \text{---} \langle 0 \rangle \\ \text{---} 1 \text{---} \langle 1 \rangle \\ \text{---} 1 \text{---} \langle 100 \rangle \\ \text{---} 1 \text{---} \langle 10000 \rangle \end{array} \end{array}$$

are both harder to predict than the previous case, and those on the right are harder than those on the left because the distances between values are larger. We can further increase that uncertainty without bound with ever more complicated Kinds.

This raises three questions. First, how should we make predictions in the face of uncertainty? Second, how should we quantify the degree of uncertainty that we face? And third, how should our decisions be affected by the degree of uncertainty? Here we will focus on the first question, touching only briefly on the other two, but rest assured we will consider all three as we proceed.

The goal of this section is to define a baseline best prediction for the value of an FRP. We express this prediction through the *price* that we would pay to receive the FRP's payoff. Using prices to describe a prediction has a long tradition. The price of a stock, for instance, is (in theory) the market's prediction of the long-run value of each share of the company. When a sports team signs a contract for a player, they are predicting how much revenue (explicitly and implicitly through championships, merchandise, advertising, et cetera) that player will bring to the organization. When an insurance company offers insurance against an event, such as damage to one's home, the price of the insurance premiums reflects the company's prediction about how much they will have to pay out.⁶⁷

The last example has a resemblance to what we face with FRPs. An insurance company makes their money *in the aggregate*. An individual homeowner's policy may or may not require a payout, but with good predictions, the company can price the premium to make a profit on a large collection of policies.⁶⁸

Similarly, with one particular FRP, we can get any of its possible payoffs, with a large enough collection of FRPs of the same Kind, we will see all of its possible payoffs in proportions close to the weights. We can thus control our gains and losses in the aggregate with the choice of price. An important implication is that our prediction about an FRP's value depends only on the *Kind* of the FRP. In effect, we are making predictions about Kinds.⁶⁹

Our best prediction of an FRP's value will be represented by the **risk-neutral price** for an FRP of that Kind, to be defined below. Here is the setup.

1. We have through the FRP Warehouse an unlimited collection of FRPs of any Kind.
2. We purchase some number of FRPs of Kind k , paying a price $\$c_k$ per unit, and our total payoff is the sum of the values of all the purchased FRPs.⁷⁰
3. We can *borrow without interest* as much money as we like to purchase FRPs, but when their payoffs are revealed, we must immediately pay back that loan.

The first task is to understand how the choice of price c_k affects what we gain or lose

⁶⁷And the companies have armies of analysts, called actuaries, whose job is to make those predictions based on the available data.

⁶⁸This assumes that the different policies are close to *independent*; if all of the homes are hit by the same hurricane, the company will lose.

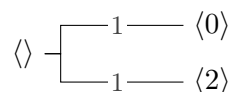
⁶⁹With a conditional constraint c from partial information about an FRP X , we get a new FRP $X \mid c$ and predictions of its value depend on its Kind $\text{kind}(X \mid c)$.

⁷⁰Remember that negative payoffs means that we have to pay out.

in the aggregate. To begin, we focus exclusively on *scalar* (1-dim) FRPs.

Consider the simplest, non-trivial FRP: *constants*, with Kinds of the form $\langle \rangle \text{ --- } 1 \text{ --- } \langle v \rangle$ for some number v . We know with *certainty* that this will payoff $\$v$. If you pay less than $\$v$, you will make a profit on each FRP you purchase, so you would purchase as many as possible. Of course, the market knows this as well, so they would not sell such an FRP for less than $\$v$. If you pay more than $\$v$, you will lose money on each FRP you purchase. Of course, you know this, so you would not buy any at such a price. For all practical purposes, this FRP is equivalent to $\$v$.

Consider next simple FRPs with Kind



Use the `frp market` application⁷¹ to purchase collections of these at different prices. With the `buy` task, you specify how many FRPs you want to buy at each of one or more prices, and the Kind. It shows your net payoff (total and per unit) for the batch purchased at each price. Here are some examples with some sample output.

⁷¹Remember, this is still your free trial, so no money changes hands yet.

```
mkt> buy 1_000_000 each @ 0.5, 0.9, 0.99, 0.999, 0.9999, 1, 1.01
...> with kind (<> 1 <0> 1 <2>).
```

Buying 1,000,000 FRPs with kind (<> 1 <0> 1 <2>) at each price

Price/Unit	Net Payoff	Net Payoff/Unit
\$0.50	\$ 500,670.00	\$ 0.500670
\$0.90	\$ 11,534.00	\$ 0.011534
\$0.99	\$ 537.00	\$ 0.000537
\$0.999	\$ -1,990.00	\$-0.001990
\$0.9999	\$ 753.00	\$ 0.000753
\$1.00	\$ -512.00	\$-0.000512
\$1.01	\$ -8,796.00	\$-0.008796

```
mkt> buy 10_000_000 each @ 0.5, 0.9, 0.99, 0.999, 0.9999, 1, 1.01
...> with kind (<> 1 <0> 1 <2>).
```

Buying 10,000,000 FRPs with kind (<> 1 <0> 1 <2>) at each price

Price/Unit	Net Payoff	Net Payoff/Unit
\$0.50	\$ 5,003,478.00	\$ 0.500348

\$0.90	\$	997,792.00	\$ 0.099779
\$0.99	\$	102,930.00	\$ 0.010293
\$0.999	\$	2,311.00	\$ 0.000231
\$0.9999	\$	96.00	\$ 0.000010
\$1.00	\$	-2028.00	\$-0.000203
\$1.01	\$	-99,224.00	\$-0.009922

```
mkt> buy 100_000_000 each @ 0.5, 0.9, 0.99, 0.999, 0.9999, 1, 1.01
...> with kind (<> 1 <0> 1 <2>).
```

Buying 100,000,000 FRPs with kind (<> 1 <0> 1 <2>) at each price

Price/Unit	Net Payoff	Net Payoff/Unit
\$0.50	\$49,995,392.00	\$ 0.499953
\$0.90	\$ 9,976,452.00	\$ 0.099765
\$0.99	\$ 1,005,452.00	\$ 0.010055
\$0.999	\$ 103,884.00	\$ 0.001039
\$0.9999	\$ 24,664.00	\$ 0.000247
\$1.00	\$ 262.00	\$ 0.000003
\$1.01	\$ -998,284.00	\$-0.009983

```
mkt> buy 1_000_000_000 each @ 0.5, 0.9, 0.99, 0.999, 0.9999, 1, 1.01
...> with kind (<> 1 <0> 1 <2>).
```

Buying 1,000,000,000 FRPs with kind (<> 1 <0> 1 <2>) at each price

Price/Unit	Net Payoff	Net Payoff/Unit
\$0.50	\$499,980,344.00	\$ 0.499803
\$0.90	\$ 99,964,150.00	\$ 0.099964
\$0.99	\$ 9,939,632.00	\$ 0.009940
\$0.999	\$ 1,001,286.00	\$ 0.001001
\$0.9999	\$ 80,598.00	\$ 0.000081
\$1.00	\$ -38,850.00	\$-0.000039
\$1.01	\$-10,088,852.00	\$-0.100889

Although it is not perfectly clearcut, there is a pattern here. When the price is low, the net payoff tends to be large and positive. The net payoff shrinks as the price approaches 1, becoming more and more negative beyond that. The third column

makes it easier to see the common pattern because the numbers across runs are all on the same “per unit scale.”

Let’s zoom in near 1 to get a closer look:

```
mkt> buy 1_000_000_000_000 each @
...>      0.9999, 0.99999, 1.00, 1.00001, 1.0001
...>      with kind (<> 1 <0> 1 <2>).
```

Buying 1,000,000,000,000 FRPs with kind (<> 1 <0> 1 <2>) at each price
(Due to large numbers, the values below may be slightly approximate.)

Price/Unit	Net Payoff	Net Payoff/Unit
\$0.9999	\$ 101,695,118	\$ 1.01695e-4
\$0.99999	\$ 8,953,455	\$ 8.95346e-6
\$1.00	\$ 915,029	\$ 9.15030e-7
\$1.00001	\$ -10,020,272	\$ -1.00203e-5
\$1.0001	\$ -99,822,037	\$ -9.9822e-5

The pattern seems similar, and it appears that 1 is the inflection point. We can guess that \$1 is the right price! And we’ll see below how this might match our intuition that 1 is the midpoint between two evenly weighted values 0 and 2.

If you were offered a price of < 1 for FRPs of the previous two example Kinds, the interest-free loan from the FRP Warehouse would let you make a profit. We give such prices a name.

Definition 18. If X is a scalar FRP, an **arbitrage price** for X is a number c such that if you pay $\$c$ per FRP, you can purchase a collection of FRPs of Kind equivalent to $\text{kind}(X)$ and *guarantee* a profit with essential certainty.

If we pay an arbitrage price for any particular number of FRPs, we can still lose money. But if we buy *enough* FRPs of the same Kind, the possibility of losing money becomes like the possibility of your air all gathering in the corner of the room. A profit is essentially guaranteed. This matches the pattern we saw in the above examples: any price less than 1 for FRPs of those Kinds is an arbitrage price. *If* we were offered an arbitrage price to purchase FRPs, we would jump on the deal and purchase as many as possible.

Arbitrage prices have an important property. If c is an arbitrage price for X and $c' < c$, then c' is also an arbitrage price for X . When c is a price that essentially

guarantees a profit, then paying a *smaller* price only makes it easier to make a profit, and this smaller price is then also an arbitrage price. In fact, we can make a stronger statement:

Property A. If c is an arbitrage price for X , then there is some real number $\epsilon > 0$ so that every $c' < c + \epsilon$ is also an arbitrage price for X .

This looks a little more mysterious at first but is based on similar intuition. If we can make an essentially guaranteed profit at some price c , then we can very, very slightly increase the price and still make a profit. The increase might be tiny indeed – ϵ can be arbitrarily small – but we can always find a higher arbitrage price.

Be careful not to conclude from Property A that we can always find arbitrage prices that are arbitrarily large. Suppose 0.99 is an arbitrage price for a particular FRP. Property A tells us that we can find an arbitrage price that is slightly bigger than 0.99. Suppose then that values < 0.9901 are arbitrage prices. Then Property A tells us that we can find a value slightly bigger than 0.9901 that is also an arbitrage price. Suppose then that values < 0.99010001 are also arbitrage prices. We can continue in this way getting larger arbitrage prices, but the amount of increase at each step can get smaller and smaller. We might *never* reach 1, for instance. In most cases of interest, the set of arbitrage prices is bounded from above, and that is how we define the risk-neutral price.

Definition 19. If X is a scalar FRP, the **risk-neutral price** for X is the smallest value r that is bigger than every arbitrage price for X .

If every finite c is an arbitrage price for X , the risk-neutral price is ∞ . If no finite c is an arbitrage price, the risk-neutral price is $-\infty$.

If we pay the risk-neutral price for the FRPs, then we might make a profit or a loss, no matter how many FRPs we purchase. There are no guarantees. Remember that a positive price means that we pay to get the FRP; a negative price means that we are paid to take it.

The set of arbitrage prices for an FRP contains every number from $-\infty$ up to **but not including** the risk-neutral price r . No reasonable person would offer us an arbitrage price to purchase FRPs because it would (essentially) guarantee them a loss. Nor would you accept an offer to pay *more* than the risk-neutral price, for it

would (essentially) guarantee you a loss. But at the risk-neutral price, neither buyer nor seller has the advantage.

The term *risk-neutral* here means that the price accounts only for typical payoff not for the magnitude of the losses that we risk. Consider FRPs with Kinds shown in Figure 44: all three have the same risk-neutral price of 10. Most of us facing a choice among these three payoffs would *not* be indifferent among them. The first guarantees a \$10 payoff. The second offers the possibility of a slightly higher payoff (\$11) at the small risk of losing \$1000 – a non-trivial loss. The third offers a bigger payoff with higher risk (losing \$10,000 is nothing to sneeze at). While you would pay \$10 for the first FRP, you would likely pay *less* for the latter two to account for the risk you face. Real betting markets account for this risk and tend to clear at prices lower than the risk neutral price. This risk matters in practice because you have limited funds available.

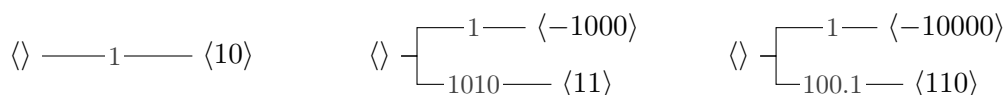
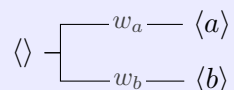


FIGURE 44. FRP Kinds with the same risk-neutral price. Are you indifferent to which of these payoffs you get?

But in our setup, at any price below the risk-neutral price, risk is not a consideration because you have unlimited funds available and can purchase an arbitrarily large number of FRPs. As such, the risk can be hedged away, and no premium for risk is needed. This pushes the equilibrium to the risk-neutral price.

Puzzle 47. Assuming $a < b$, can the risk-neutral price of the FRP with Kind



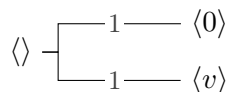
be less than a ? Greater than b ? Why or why not?

Can it be equal to a or b ? (Remember $w_a, w_b > 0$.) Why or why not?

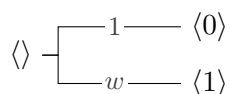
If w_b is very much bigger than w_a , do you expect the risk-neutral price to be closer to a or to b ?

Activity. Empirically evaluate the risk-neutral price of several scalar FRPs, using the `buy` command as above. As a starting point, consider FRPs with a few simple Kinds, like:

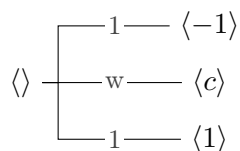
1. For various values of v ,



2. For various values of w ,



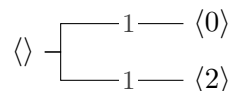
3. For various values of $-1 < c < 1$ and of w , starting with $w = 1$,



Try some other examples. Can you guess the relationship between an FRP's Kind and its risk-neutral price? Don't forget the results of our earlier demos. What do you expect to see when you tabulate the values of a large sample of FRPs of the same Kind? What does this mean for the risk-neutral price?

7.1 Fundamental Properties of Risk-Neutral Prices

The risk-neutral price of an FRP X represents a good prediction of X 's value. This statement requires some interpretation. For instance, we saw the FRP with Kind



has a risk-neutral price of 1. If this is to be considered a good prediction of the FRP's value, shouldn't it be a concern that *1 is not a possible value of X* ? If you predict 1, you will *always* be wrong. This is true, but a key point is that there are

different ways to assess the accuracy of prediction. If we require that we guess the exact value, then it is easy to construct Kinds and “good” predictions that are almost always wrong and almost always far from the true value. For example, with the Kind `uniform(1, 2, ..., 10_000_000_000)`, always guessing 10,000,000,000 does as well as possible in guessing the exact value but is wrong and very far from the true value the vast majority of the time. While it is sometimes sensible to prioritize guessing the exact value, that turns out not to be the most useful criterion in practice.

The market captures a different notion of prediction: prediction accuracy *in the aggregate*. By defining predictions in terms of prices, our predictions are fine-tuned to the structure of the Kind. If you guess below the risk-neutral price, the value will tend to be above your guess (and you can make money almost certainly in the market). If you guess above the risk-neutral price, the value will tend to be below your guess (and you will lose money in the market if you purchase enough). The risk-neutral price is thus a “typical” value of X ; it gives us a prediction for that value that is *as close possible to that value* in a particular sense to be described below.

We can learn quite a lot about X from its risk-neutral price. If X is the *constant* FRP with value v , we know its risk-neutral price is v .

If X and Y are FRPs and we *know* that the value of X will be \leq the value of Y , we write $X \leq Y$. In this case, if c is an arbitrage price for X , it must also be an arbitrage price for Y as our payoff with Y will be at least that with X . Hence, X ’s *risk-neutral price is $\leq Y$ ’s risk-neutral price*. For instance, if all the values of X are between a and b , with $a \leq b$, then X ’s risk-neutral price must be between a and b by applying this fact to X and the constant FRPs at a and b .

For a real-number s , we write sX for the transformed FRP that *scales the value of X , whatever it is, by s* . Formally, this is $\psi_s(X)$ where ψ_s is the statistic defined by $\psi_s(x) = sx$, which just scales its argument by s . If we know the risk-neutral price for X , can we find the risk-neutral price for sX ? Consider a large batch of FRPs with Kind equivalent to `kind(X)`: $X_{[1]}, \dots, X_{[m]}$. In the market, we can choose to purchase the transformed batch $sX_{[1]}, \dots, sX_{[m]}$. If c is any arbitrage price for X , then for large enough m , purchasing the batch $X_{[1]}, \dots, X_{[m]}$ at unit price $\$c$ would essentially guarantee us a profit. If $s > 0$, then we would also get a profit from the same batch with payoffs $sX_{[1]}, \dots, sX_{[m]}$ at unit price $\$sc$ because the payoffs and the profit are all just scaled by s . So, sc is an arbitrage price for sX . (If $s < 0$, we just use the same argument scaling by $-s$, yielding the same result.) Hence, *the*

risk-neutral price of sX is just s times the risk-neutral price for X .

We can (and will) go on like this, *deriving properties of the risk-neutral price from the logic of arbitrage prices*. But first it will be nice to have a ... crisper notation.

Notation. If X is an FRP, we will use $\mathbb{E}(X)$ to denote the risk-neutral price of the FRP. (In the playground, this is $\mathbb{E}(X)$.)

Consider one more property of risk-neutral prices. Let $d = \dim(X)$ and define $Y = \text{proj}_i(X)$ and $Z = \text{proj}_j(X)$ for some $1 \leq i, j \leq d$. If ψ is that statistic of type $d \rightarrow 1$ defined by $\psi(x) = x_i + x_j$, we write $Y + Z$ to mean $\psi(X)$. This gives us three FRPs derived from X : Y , Z , and $Y + Z$.

We can find $\mathbb{E}(Y + Z)$ from $\mathbb{E}(Y)$ and $\mathbb{E}(Z)$. If $c_1 \leq \mathbb{E}(Y)$ and $c_2 < \mathbb{E}(Z)$ be arbitrage prices, then $c_1 + c_2$ is an arbitrage price for $Y + Z$ because we can make arbitrarily large amounts of money from the Z payoffs even if we lose a little from the Y payoffs. Similarly, if $c_1 < \mathbb{E}(Y)$ and $c_2 \leq \mathbb{E}(Z)$, $c_1 + c_2$ is an arbitrage price for $Y + Z$. But $\mathbb{E}(Y) + \mathbb{E}(Z)$ *cannot* be an arbitrage price for $Y + Z$ because the payoff at this price from any batch of $Y + Z$ clones is equivalent to a payoff from batches of Y 's and Z 's at their risk-neutral price, for which a profit is *not* essentially guaranteed. It follows that $\mathbb{E}(Y) + \mathbb{E}(Z)$ *is* the risk-neutral price for $Y + Z$. By Bob's equation from Section 6 $\text{Sum}(a :: b) = \text{Sum}(\langle \text{Sum}(a), \text{Sum}(b) \rangle)$, this generalizes to any number of components.

Together, these arguments give us four key properties of risk-neutral prices.

Box 20. Key Properties of Risk-Neutral Prices. Let X, Y be FRPs.

Constancy. If X is a constant FRP with value v ,

$$\mathbb{E}(X) = v. \quad (7.9)$$

Scaling. For any real number s ,

$$\mathbb{E}(sX) = s \mathbb{E}(X). \quad (7.10)$$

Ordering. If $X \leq Y$,

$$\mathbb{E}(X) \leq \mathbb{E}(Y). \quad (7.11)$$

Additivity. If X_1, X_2, \dots, X_n are FRPs of common dimension,

$$\mathbb{E}(X_1 + X_2 + \dots + X_n) = \mathbb{E}(X_1) + \mathbb{E}(X_2) + \dots + \mathbb{E}(X_n). \quad (7.12)$$

In the condition for Scaling, $sX = \zeta(X)$ for statistic $\zeta(x) = sx$. In the condition for Ordering, $X \leq Y$ means that the value of X is *known* to be \leq the value of Y . In the condition for Additivity, for *any* FRPs X_1, \dots, X_n of common dimension, we can always build an FRP X for which $X_1 + \dots + X_n = \psi(X)$ and $X_i = \varphi_i$, for some statistics ψ and φ_i , $i \in [1 \dots n]$.

These properties provide critical tools for finding and working with risk-neutral prices. Even with the formula for risk-neutral prices that we derive in the next subsection, it will often more direct to solve or simplify a problem with the logic of these properties. In practice, we often use the Constancy, Scaling, and Additivity properties together to establish that

$$\mathbb{E}(\alpha_0 + \alpha_1 X_1 + \alpha_2 X_2 + \dots + \alpha_n X_n) = \alpha_0 + \alpha_1 \mathbb{E}(X_1) + \alpha_2 \mathbb{E}(X_2) + \dots + \alpha_n \mathbb{E}(X_n), \quad (7.13)$$

for constants $\alpha_0, \alpha_1, \dots, \alpha_n$. This derived property that is called **linearity**.

We derived arbitrage prices – and thus the risk-neutral price – for a scalar FRP by considering an arbitrarily large collection of FRPs with the same Kind. It follows that the risk-neutral price of an FRP is *determined by its **Kind*** not by the FRP's particular value.

All FRPs with the same Kind have the same risk-neutral price. The risk-neutral price of an FRP is thus a property of the Kind and does not depend on any particular FRPs produced value.

Up to now in this section, we have been considering the risk-neutral prices of scalar FRPs. How do we handle FRPs of dimension > 1 ? Because the value of such an FRP is a tuple, there is not a single payoff, so it is unclear how to assign a single risk-neutral price to it. Instead, we extend the risk-neutral price to a more general idea – the **expectation**. If X is a scalar FRP, its expectation $\mathbb{E}(X)$ is its risk-neutral price. If $\dim(X) > 1$, its expectation $\mathbb{E}(X)$ is the tuple whose components are the risk-neutral prices of X 's scalar component FRPs.

Definition 21. Let X be an FRP. We use $\mathbb{E}(X)$ to denote the **expectation of X** .

- If $\dim(X) = 1$, $\mathbb{E}(X)$ is just the risk-neutral price of X .
- If $\dim(X) = n > 1$ and FRPs $\langle X_1, X_2, \dots, X_n \rangle$ are its scalar components, then $\mathbb{E}(X)$ is an n -tuple of numbers defined by

$$\mathbb{E}(X) = \langle \mathbb{E}(X_1), \mathbb{E}(X_2), \dots, \mathbb{E}(X_n) \rangle, \quad (7.14)$$

that is, the tuple containing the risk-neutral prices of the components.

Note that the expectation of an FRP has the same dimension as the FRP.

We often write expectations in terms of FRPs, but it is fine to refer to the expectation of a Kind because $\mathbb{E}(X)$ is a property of $\text{kind}(X)$ and does not depend on X 's specific value.

The good news is that because the expectation is derived from risk-neutral prices, **all the properties 7.9, 7.10, 7.11, and 7.12 continue to hold for any FRP**. We can view equations (7.10), (7.11), and (7.12) as telling us that \mathbb{E} maps the operations of scaling, ordering, and adding on FRPs/Kinds to the analogous operations on values. These operations all work for numbers. For numbers x and y , we can scale them sx , order them $x \leq y$, and add them $x + y$. And the operations also work for *tuples of numbers* of a common dimension n ;⁷² we can scale, order, and add them:

$$s\langle x_1, x_2, \dots, x_n \rangle = \langle sx_1, sx_2, \dots, sx_n \rangle \quad (\text{Scaling})$$

$$\langle x_1, x_2, \dots, x_n \rangle \leq \langle y_1, y_2, \dots, y_n \rangle \text{ if and only if } x_1 \leq y_1 \wedge \dots \wedge x_n \leq y_n \quad (\text{Ordering})$$

$$\langle x_1, x_2, \dots, x_n \rangle + \langle y_1, y_2, \dots, y_n \rangle = \langle x_1 + y_1, x_2 + y_2, \dots, x_n + y_n \rangle. \quad (\text{Additivity})$$

For instance, Additivity (7.12) with $\psi_i = \text{proj}_i$ and equation (7.14) imply that

$$\mathbb{E}(\text{Sum}(X)) = \text{Sum}(\mathbb{E}(X)) \quad (7.15)$$

for the **Sum** statistic that sums the components of its argument.

A frequently occurring special case of equation (7.14) is when X is an *independent mixture* of X_1, X_2, \dots, X_n scalar FRPs:

$$\mathbb{E}(X_1 \star X_2 \star \dots \star X_n) = \langle \mathbb{E}(X_1), \mathbb{E}(X_2), \dots, \mathbb{E}(X_n) \rangle. \quad (7.16)$$

⁷²See Section F.9.3, which discusses “vector” operations on tuples.

Independence tells us that we can price the components separately.

Note also that when we apply a conditional constraint to an FRP X , we get another FRP $X \mid c$, which consequently has expectation $\mathbb{E}(X \mid c)$ and Kind $\text{kind}(X \mid c) = \text{kind}(X) \mid c$. Because $X = X \mid \top$, $\mathbb{E}(X)$ is the same as $\mathbb{E}(X \mid \top)$.

Puzzle 48. Given that we observe an FRP X to have value 4, what is its expectation (i.e., risk-neutral price)? We would write this with the conditional constraint as $\mathbb{E}(X \mid X = 4)$.

The following examples illustrate how we can use the properties above to find expectations.

Example 7.1. If X has Kind of the form

$$\langle \rangle \text{ --- } \begin{cases} a \text{ --- } \langle -1 \rangle \\ b \text{ --- } \langle 1 \rangle \end{cases}$$

what is $\mathbb{E}(X^2)$? (Recall that X^2 is the “inline” notation for the transformed FRP $\psi(X)$ with the simple statistic $\psi(x) = x^2$; see the discussion of inlined statistics on page 47. It is *not* the same as $X \star X = X \star \star 2$.)

When X produces a value v , that value is fed to the input port of X^2 through an adapter that outputs v^2 . Here, v can be either -1 or 1, and in both cases, $v^2 = 1$. This means that 1 is the only possible value of X^2 . It is constant.

Hence, by the Constancy property (7.9), $\mathbb{E}(X^2) = 1$.

Example 7.2 Changing Units

The FRP D represents a measured distance in kilometers, and we’d like to create an FRP X that represents the same distance in *miles*. This transformation is straightforward with the statistic $\psi(d) = cd$ for a constant $c \approx 0.621371$. So, $X = \psi(D)$, though for such a simple transformation we most often *inline* the statistic, writing it as $X = cD$.

The Scaling property (7.10) tells us that $\mathbb{E}(X) = \mathbb{E}(cD) = c\mathbb{E}(D)$. The expectation scales by the same factor.

Example 7.3. Let A be an FRP representing a random angle, measured in degrees. Even without knowing $\text{kind}(A)$, we know that its values must lie in the interval $[0_360)$. If A' is an FRP derived from A by converting its values to radians, then its values must lie in the interval $[0_2\pi)$. Let $X = \cos(A')$ and $Y = \sin(A')$ represent the cosine and sine of this angle; their values each lie in the interval $[-1_1]$.

The Ordering property (7.11) of expectations tells us that

$$\begin{aligned} 0 &\leq \mathbb{E}(A) \leq 360 \\ 0 &\leq \mathbb{E}(A') \leq 2\pi \\ -1 &\leq \mathbb{E}(X) \leq 1 \\ -1 &\leq \mathbb{E}(Y) \leq 1. \end{aligned}$$

Example 7.4 Measuring Uncertainty

The risk-neutral price of a scalar FRP is a prediction of the FRP's value. The greater the uncertainty in the Kind of that FRP, the harder it is to accurately predict the FRP's value. However, even when the uncertainty is high, our prediction might by chance be close to the actual value, a lucky guess so to speak. So how do we quantify the uncertainty in a Kind of FRPs?

The answer is to construct a transformed FRP with a customized statistic that describes how the value deviates from our prediction. Here, we will introduce three ways to quantify uncertainty that are called the *Mean Absolute Deviation*, the *Variance*, and the *Entropy*. We will focus here only on 1-dimensional FRPs.

Let Y be a scalar FRP with $K = \text{kind}(Y)$ its Kind and $\mu = \mathbb{E}(Y)$ its risk-neutral price. If v is a possible value of Y , we will write $K(v)$ for the weight on that value in the canonical form of Y 's Kind.

First, define a family of statistics φ_c , indexed by real numbers c , where

$$\varphi_c(x) = |x - c|. \quad (7.17)$$

This statistic measures how far its argument value is from the number c . We

First of the "Measuring Uncertainty" example series.

define the *Mean Absolute Deviation* of Y , denoted $\text{MAD}(Y)$, to be

$$\text{MAD}(Y) = \mathbb{E}(\varphi_\mu(Y)). \quad (7.18)$$

This is *our prediction of how far Y 's value is from our prediction of its value*. That is, by the nature of the statistic φ_μ , the expectation of the FRP $\varphi_\mu(Y)$ measures the deviation between Y 's value and Y 's risk-neutral price.

Because $\varphi_c(x) \geq 0$ for all x , the FRP $\varphi_\mu(Y)$ can never be negative, so by the Ordering property (7.11), $\text{MAD}(Y) \geq 0$. The extreme case is when Y is a constant FRP. In this case, $\mathbb{E}(Y)$ equals the only possible value, so $\text{MAD}(Y) = 0$.

One novel feature of this is that we use a statistic that is customized to match the FRP we are transforming. While we could look at $\mathbb{E}(\varphi_c(Y))$ for any c , we use $c = \mu = \mathbb{E}(Y)$. Because φ_μ has a relatively simple form, we would tend to inline this statistic and write $\text{MAD}(Y) = \mathbb{E}(|Y - \mathbb{E}(Y)|)$.

The Mean Absolute Deviation is intuitive: it predicts how far the actual value of an FRP will be from its predicted value (risk-neutral price). The distance between value and prediction is in the same units as Y , which is nice. Unfortunately, the absolute value $|\cdot|$ in the statistic makes it harder to work with mathematically. For that reason, we introduce the *Variance*.

Define a family of statistics ψ_c , with one statistic for each real number c , where

$$\psi_c(x) = (x - c)^2. \quad (7.19)$$

Like φ_c above, this statistic measures how far its argument value is from the number c , but it measures distance in *squared* units. We define the *Variance* of Y , denoted $\text{Var}(Y)$, to be the expectation of $\psi_\mu(Y)$:

$$\text{Var}(Y) = \mathbb{E}(\psi_\mu(Y)) = \mathbb{E}((Y - \mathbb{E}(Y))^2), \quad (7.20)$$

where the second form is the inlined expression of the statistic. This is *our prediction of how far Y 's value is from our prediction of its value in squared units*. If the variance is high, we predict that Y 's value will typically be far from its expectation; if the variance is small, we predict that Y 's value will typically be close to its expectation.

The quadratic versus the absolute value is the distinction between ψ_c and φ_c and between **Var** and **MAD**. The move to squared units makes the variance's numeric values a bit harder to understand, but the quadratic makes it easier to simplify and manipulate, as we will see.

Again, because $(x-c)^2 \geq 0$ for all x , $\psi_c(Y)$ cannot be negative, so $\text{Var}(Y) \geq 0$ by the Ordering property. The extreme case is when Y is a constant FRP. In this case, $\mathbb{E}(Y)$ equals the only possible value and $\text{Var}(Y) = 0$. A constant FRP embodies no uncertainty.

Finally, define a family of statistics ζ_k , with one statistic for each canonical Kind k , where

$$\zeta_k(x) = -\lg k(x), \quad (7.21)$$

where \lg denotes the logarithm base 2 and $k(x)$ is the canonical weight associated with value x in Kind k . Unlike the previous two statistics that measure distances between values, this statistic measures the weights on each value. The statistic is *larger* for values that have *lower* weight and smaller for values that have higher weight. Remember that a canonical weight is ≤ 1 , so the log of that weight is negative and the negative sign makes the result non-negative. In other words, $\zeta_k(x) \geq 0$ for any value x in a branch of Kind k .

We define the *Entropy* of Y , denoted $\mathbb{H}(Y)$, to be

$$\mathbb{H}(Y) = \mathbb{E}(\zeta_K(Y)) = \mathbb{E}(-\lg K(Y)), \quad (7.22)$$

where the second form is the inlined expression of the statistic and $K = \text{kind}(Y)$.

Because $\zeta_K(Y)$ cannot be negative and can be at most $-\lg p_{\min} = \lg(1/p_{\min})$ where p_{\min} is the smallest weight in $\text{kind}(Y)$, the Ordering property (7.11) implies that

$$0 \leq \mathbb{H}(Y) \leq \lg(1/p_{\min}).$$

If Y is a constant FRP, then $p_{\min} = 1$, which implies that $\mathbb{H}(Y) = 0$. As with the other measures, a constant – with no uncertainty – gives the minimal value 0.

The entropy depends only on the number of values and their weights, not on the values themselves. Its value is measured in *bits*. If Y has Kind

`uniform(1, 2, ..., n)` for some positive integer n , then all the canonical weights in $\text{kind}(Y)$ equal $1/n$, so $\zeta_K(Y)$ is a constant FRP. In this case, $\mathbb{H}(Y) = \lg n$, which is within 1 of the number of bits used to represent the number n in binary (base 2). We will develop the interpretation of $\mathbb{H}(Y)$ later in this example series and even further in Chapter 7.

Example 7.5 Sums

In Section 6, Alice and Bob figured out a clever way to compute the Kind for the total of 100 die rolls, `Sum(d6 ** 100)` in the playground, where `d6` is the Kind for a roll of a balanced six-sided die. Fortunately, to compute the risk-neutral price for this, we do not need such clever tricks due to the Additivity property of equation (7.15).

In the playground, we can create the Kind `d6` and an FRP `D_6` that represents the roll of a balanced six-sided die.

```
pgd> d6 = uniform(1, 2, ..., 6)
pgd> D_6 = frp(d6)
pgd> D_6
An FRP with value <4>
```

The Kinds for multiple rolls of the dice are computed with an independent mixture power, but their sizes quickly grow large. For instance, `d6 ** 4`, `d6 ** 8`, `d6 ** 100` have respected sizes 1296, 1679616, and 653318623500070906096690267158057820537143710472954871543071966369497141477376.

We can directly create FRPs, however, to simulate large number of dice rolls without any problem:

```
pgd> Rolls4 = D_6 ** 4
pgd> Rolls8 = D_6 ** 8
pgd> Rolls100 = D_6 ** 100
pgd> Rolls4
An FRP with value <2, 1, 3, 2>
(It may be slow to evaluate its kind.)
pgd> Rolls8
```

```
An FRP with value <5, 6, 3, 4, 4, 1, 4, 3>.
```

```
(It may be slow to evaluate its kind.)
```

```
pgd> Rolls100
```

```
An FRP with value <2, 1, 3, 3, 4, 1, 1, 6, 3, 5, 5, 1, 2, 1, 3,
4, 6, 2, 2, 5, 2, 1, 4, 6, 5, 6, 3, 6, 3, 2, 6, 2, 1, 1, 5, 1,
1, 6, 3, 6, 3, 3, 3, 4, 5, 4, 3, 5, 6, 5, 5, 2, 2, 6, 2, 3, 3,
6, 3, 1, 2, 4, 4, 4, 4, 3, 1, 4, 1, 3, 5, 1, 3, 2, 3, 1, 4, 4,
2, 1, 2, 6, 3, 4, 3, 1, 5, 1, 1, 2, 5, 3, 5, 2, 4, 2, 3, 6, 3,
1>. (It may be slow to evaluate its kind.)
```

The value of `Rolls4` shows us the four rolls. This is an independent mixture, and recall that it is equivalent to `clone(D_6) * clone(D_6) * clone(D_6) * clone(D_6)`, as in equation (4.7), and similarly for `Rolls8` and `Rolls100`. The Kind of `Rolls4` has not yet been computed, as the message indicates, though we could force it to be by calling `kindn(Rolls4)`. (Try it!) Deferring the computation of the Kind when the size may be large allows us to construct FRPs even if the Kind is slow to compute.

(Note that computing a repeated independent mixture of an FRP without the `clone` is generally not what we want.

```
pgd> D_6 * D_6 * D_6 * D_6    # not quite right, all values the same
An FRP with value <4, 4, 4, 4>.
```

`D_6` has only one value fixed for all time, so however many terms in the mixture, they will all have the same value.)

Without Alice and Bob's trick, if we try to compute the risk-neutral price with the `E` operator, the playground detects that the Kind is hard to compute and approximates the answer instead.

```
pgd> Sum(Rolls100)
```

```
An FRP with value <322>. (It may be slow to evaluate its kind.)
```

```
pgd> E(Sum(Rolls100))
```

```
+-- Computing approximation (tolerance 0.01) --+
```

```
| 350.0251 |
```

```
+-----|
```

We could force the Kind to be computed with `E(Sum(Rolls100), force_kind=True)` or change the approximation tolerance with `E(Sum(Rolls100), tolerance=0.001)` if we prefer, though the former calculation would not complete given the size of the Kind.

Additivity gives us a quick and exact answer for `E(Sum(Rolls100))`. First, remember that `E(D_6) == E(clone(D_6))` because the an FRP's expectation only depends on its Kind not its particular value. And we know that

```
pgd> E(D_6)
7/2
```

Second, Definition 21 tells us that `E(D_6 ** 100)` is equal to

```
<E(clone(D_6)), E(clone(D_6)), ..., E(clone(D_6))>
```

Indeed, we can compute it directly in the playground:

```
pgd> E(D_6 ** 100)
<7/2, 7/2, 7/2, ..., 7/2, 7/2>
```

where 95 of the values have been elided for brevity. Finally, the Additivity property (equation 7.15) implies that `E(Sum(D_6 ** 100)) == Sum(E(D_6 ** 100))` and

```
pgd> Sum(E(D_6 ** 100))
350
```

which is the exact answer we want. More generally, the Additivity property tells us that for any $n \in [1..)$, $\mathbb{E}(D_6 \star n) = n \mathbb{E}(D_6) = \frac{7}{2}n$.

Indeed, combining equations (7.16) and (7.15), we have that for any FRP X

$$\mathbb{E}(\text{Sum}(X \star n)) = n \mathbb{E}(X). \quad (7.23)$$

Following up on the previous example, consider the statistic `Mean` that computes the arithmetic average of its input's components. For any value v , we have that $\text{Mean}(v) = \text{Sum}(v) / \text{dim}(v)$. Hence, combining equation (7.23) and the Scaling property (7.10) yields

$$\mathbb{E}(\text{Mean}(X \star n)) = \mathbb{E}(X). \quad (7.24)$$

The average of an independent mixture power has the same expectation as each term.

It is fairly common when taking measurements to use averages to compute a more “representative” quantity. Surveyors average multiple measurements to estimate distances. Baseball coaches use batting averages to assess hitters. Teachers take averages of students’ exams to assign grades. In light of equation 7.24, we might ask how averaging helps if the expectation – our prediction – does not change by averaging. The next example sheds light on this question.

Example 7.6 Averaging and Uncertainty

Our uncertainty about a system increases with the difficulty of making accurate predictions. Example 7.4 introduced several different ways to *quantify the uncertainty* in a Kind. We consider multiple quantities for this purpose because in some context, one might emphasize subtly different features or have a useful practical interpretation or be more convenient to work with.

Here, we consider the *variance* of a Kind and in particular the variance of an independent mixture power. As defined earlier: if Y is a scalar FRP with risk-neutral price $\mathbb{E}(Y)$, its variance is $\text{Var}(Y) = \mathbb{E}(\psi_{\mathbb{E}(Y)}(Y))$ where $\psi_c(y) = (y - c)^2$ is a family of statistics, one for each constant c . As this statistic is a simple quadratic, we often “inline” it as $\text{Var}(Y) = \mathbb{E}((Y - \mathbb{E}(Y))^2)$. The variance is our best prediction of the squared distance of Y ’s value from its expectation. When the variance is small, Y ’s value tends to be close to its expectation. When its large, Y ’s value tends to be far from its expectation. We saw in Example 7.4 that $\text{Var}(Y) \geq 0$.

To get a different perspective on the variance, we write the statistic ψ_c by expanding the quadratic: $\psi_c(y) = y^2 - 2cy + c^2$. In this form, we can apply the Constancy, Scaling, Additivity properties to get the **variance shortcut**:

$$\begin{aligned} \text{Var}(Y) &= \mathbb{E}(\psi_c(Y)) \\ &= \mathbb{E}(Y^2 - 2\mathbb{E}(Y)Y + (\mathbb{E}(Y))^2) \\ &= \mathbb{E}(Y^2) + \mathbb{E}(-2\mathbb{E}(Y)Y) + \mathbb{E}((\mathbb{E}(Y))^2) && \text{(by Additivity)} \\ &= \mathbb{E}(Y^2) + \mathbb{E}(-2\mathbb{E}(Y)Y) + (\mathbb{E}(Y))^2 && \text{(by Constancy)} \\ &= \mathbb{E}(Y^2) - 2\mathbb{E}(Y)\mathbb{E}(Y) + (\mathbb{E}(Y))^2 && \text{(by Scaling)} \end{aligned}$$

Part of the “Measuring Uncertainty” series.

See page 47 on inlining.

$$= \mathbb{E}(Y^2) - (\mathbb{E}(Y))^2. \quad (7.25)$$

Remember that the expectation $\mathbb{E}(Y)$ is just a number, so when we apply the Scaling property to $\mathbb{E}(-2\mathbb{E}(Y)Y)$, we pull out the constant $-2\mathbb{E}(Y)$. We called this combined application of Constancy, Scaling, and Additivity *linearity* in equation (7.13). The variance shortcut reveals the variance as the difference between squaring after and before computing the risk-neutral price. There is no difference if Y is a constant FRP, but as the possible values of Y become more widely spread, squaring will spread them further, increasing the expectation of Y^2 relative to the square $(\mathbb{E}(Y))^2$ of the original price.

With these ideas in hand, we will investigate the question of why we average measurements. In this setting, $Y = \text{Mean}(X \star n)$ for a scalar FRP X and a positive integer n . X and its clones represent individual measurements of some quantity (e.g., one surveyor's distance, one batter's at bat, one exam), and Y is the average of n such measurements. We saw in (7.24) that $\mathbb{E}(Y) = \mathbb{E}(X)$. The expectation of the FRP representing the average is the same as for the FRP representing a single measurement. What can we say about the uncertainty in $\text{kind}(Y)$ versus $\text{kind}(X)$?

In this example, we will empirically investigate this question in the playground, using the built-in `Var` operator. We will start with a couple arbitrary Kinds; you can do these steps with other Kinds as well.

There is also a *statistic* `Variance` that is different; it computes the “sample variance” of its input value's components.

```
pgd> k1 = geometric(1, 2, 3, 4, 5)
pgd> k2 = Mean(k1 ** 2)
pgd> k4 = Mean(k1 ** 4)
pgd> k6 = Mean(k1 ** 6)
pgd> k8 = Mean(k1 ** 8)    # this might take a few moments
pgd> Var(k1)
0.5837669094693028
pgd> Var(k2) / Var(k1)
0.5
pgd> Var(k4) / Var(k1)
0.25
```

```
pgd> Var(k6) / Var(k1)
0.166666666666666666666666666665
pgd> Var(k8) / Var(k1)
0.125
```

When we average 2, 4, 6, and 8 independent copies, the variance decreases by factors of 1/2, 1/4, 1/6, and 1/8. Let's try it with a different Kind to start

```
pgd> k1 = either(0, 1)
pgd> k3 = Mean(k1 ** 3)
pgd> k5 = Mean(k1 ** 5)
pgd> k10 = Mean(k1 ** 10)
pgd> k16 = Mean(k1 ** 16)    # this might take a moment
pgd> Var(k1)
1/4
pgd> Var(k3) / Var(k1)
0.33333333333333333333333333333
pgd> Var(k5) / Var(k1)
0.2
pgd> Var(k10) / Var(k1)
0.1
pgd> Var(k16) / Var(k1)
0.0625
```

This shows the same pattern! So, we hypothesize that

$$\text{Var}(\text{Mean}(X \star n)) = \frac{1}{n} \text{Var}(X). \quad (*)$$

Puzzle 49. Alice and Bob's Monoidal statistic trick is embodied in the playground function `fast_mixture_pow`, and `Mean` is just `Sum` (a monoidal statistic) divided by the number of terms. Use these facts to check our hypothesis (*) for larger values of n , like 100, 500, or 1000 with at least the Kind `either(0,1)`.

It turns out that our hypothesis is true. Although our highly suggestive evidence here does not firmly establish this claim, we will see an argument later that proves it. For now, consider how that fact addresses our question – why do we average? When we compute $\mathbb{E}(\text{Mean}(X \star n))$ we want to predict the same value we are measuring in any single copy of X , so it makes sense that $\mathbb{E}(\text{Mean}(X \star n)) = \mathbb{E}(X)$. The average measurement should be measuring the same underlying quantity.

$\text{Var}(\text{Mean}(X \star n)) = \frac{1}{n} \text{Var}(X)$ tells us that *averaging reduces the uncertainty in our prediction* of that quantity by a factor proportional to the number of measurements, when the repeated measurements are all independent. Lower uncertainty means that our predictions tend to get more accurate. This is why we average.

Example 7.7 Aces

Here we revisit Alice’s deck of cards from Section 6. Let S be the FRP of dimension 52 (and size 52!) whose values are all 52! permutations of $1, 2, \dots, 52$. Let $\mathcal{A} = \{1, 14, 27, 40\}$ be the set of values in $[1..52]$ that correspond to the four aces in the deck.

We can use the FRP S as a model of equally-weighted shuffles of a standard deck of cards and model components in \mathcal{A} to be aces. We want to use this model to predict how many cards are between each ace in the deck.

Define a statistic ψ that takes a value of S and returns $\langle a_1, a_2, a_3, a_4, a_5 \rangle$, where a_1 is number of components of S ’s value *before* the first ace in the deck (a value in \mathcal{A}); a_2 is the number of components strictly between the first two aces in the deck (values in \mathcal{A}); a_3 is the number of components strictly between the second and third aces in the deck (values in \mathcal{A}); a_4 is the number of components strictly between the third and fourth aces in the deck (values in \mathcal{A}); and a_5 is the number of components strictly *after* the final ace in the deck (value in \mathcal{A}).

Puzzle 50. Show how to define the statistic ψ in the playground. Call it `between_aces`.

We can use the `shuffle` FRP factory to create S :

```
pgd> S = shuffle(irange(52))    # Values are permutations of 1..52
pgd> S
```

```
An FRP with value <15, 48, 16, 41, 20, 22, 29, 4, 17, 34, 39,
14, 26, 8, 49, 1, 18, 2, 31, 32, 52, 37, 36, 42, 19, 3, 35, 28,
50, 25, 38, 33, 45, 24, 44, 46, 11, 43, 7, 23, 9, 13, 12, 40,
30, 47, 10, 21, 27, 6, 51, 5>.
```

We can apply the statistic you defined in the puzzle to look at the gaps between aces in the simulated deck.

```
pgd> A = between_aces(S)
pgd> A
An FRP with value <11, 3, 27, 4, 3>. (It may be slow to evaluate its kind.)
```

The FRP A has dimension 5, and we can write its components as A_1, A_2, A_3, A_4, A_5 whose values have the meaning described earlier. We can look at the components individually, for instance

```
pgd> A[1]
An FRP with value <11>. (It may be slow to evaluate its kind.)
pgd> A[3]
An FRP with value <27>. (It may be slow to evaluate its kind.)
```

and similarly for the other aces.

To start, we would like to compute $\mathbb{E}(A_1 + A_2 + A_3 + A_4 + A_5)$. The FRP here is transform of A , which is in turn a transform of S . The result is a transform of S by statistic $\zeta = \text{Sum} \circ \text{between_aces}$ (“Sum after between_aces”), which can be expressed in multiple equivalent forms in the playground:

```
pgd> Sum(A)
pgd> Sum(between_aces(S))
pgd> S ^ Sum(between_aces)
pgd> S ^ between_aces ^ Sum
```

These are all the same FRP. (Make sure you understand why.)

Puzzle 51. In the playground, construct the statistic `Sum(between_aces)`. This is the *composition* of the two statistics: to apply it, we first apply `between_aces` to a deck and then apply `Sum` to the value it returns. “Sum after `between_aces`.”

Evaluate `Sum(between_aces(S))`. If you clone S and do this again, what possible values might you see?

Try:

```
pgd> psi = Sum(between_aces)
pgd> FRP.sample(100, psi(S))
```

What do the results tell you? Do the result change if you demo 1000 samples? Can you explain these tables?

The results of the previous puzzle strongly suggest that the statistic ζ is an *invariant*: it gives the same value for every possible shuffle of the deck. We can confirm that empirical result with logic: for every shuffle s , $\zeta(s)$ counts all the cards in the deck *except the aces*. So $\zeta(s) = 48$, and thus $\zeta(S) = A_1 + A_2 + A_3 + A_4 + A_5 = 48$ is a constant FRP. By the Constancy property (7.9), we therefore have that $\mathbb{E}(\zeta(S)) = 48$.

The Additivity property (7.12) now applies:

$$\begin{aligned} 48 &= \mathbb{E}(\zeta(S)) \\ &= \mathbb{E}(A_1 + A_2 + A_3 + A_4 + A_5) \\ &= \mathbb{E}(A_1) + \mathbb{E}(A_2) + \mathbb{E}(A_3) + \mathbb{E}(A_4) + \mathbb{E}(A_5), \end{aligned}$$

which constrains the expectations of the counts for the individual aces. With a bit more logic, we can go even further. We can show that A_1 , A_2 , A_3 , A_4 , and A_5 have the *same Kind*, and thus the same expectation. By the Additivity

property (7.12), it follows that

$$\begin{aligned}\mathbb{E}(A_1 + A_2 + A_3 + A_4 + A_5) &= \mathbb{E}(A_1) + \mathbb{E}(A_2) + \mathbb{E}(A_3) + \mathbb{E}(A_4) + \mathbb{E}(A_5) \\ &= 5\mathbb{E}(A_1)\end{aligned}$$

so,

$$\mathbb{E}(A_1) = \mathbb{E}(A_2) = \mathbb{E}(A_3) = \mathbb{E}(A_4) = \mathbb{E}(A_5) = 9.6$$

And we have found the expectations of the A_i 's.

We will establish that the A_i 's have the same Kind by a direct argument and then illustrate it in the playground with a smaller “deck” where we can see the trees.

First, consider A_1 and A_4 , though our argument will apply to any $i \neq j$ with $i, j \in [1..5]$. Let \mathcal{S} be the set of all permutations $\langle 1, 2, \dots, 52 \rangle$, i.e., possible values of the FRP. Define a function cut_{14} that maps elements of \mathcal{S} to elements of \mathcal{S} (denoted $\text{cut}_{14}: \mathcal{S} \rightarrow \mathcal{S}$) that exchanges the positions of cards strictly before the first ace in s and the cards strictly between the third and fourth aces without changing the order of the cards within each group. Notice that if we apply cut_{14} to a shuffle s and then apply cut_{14} again, we get the original shuffle back! That is, $\text{cut}_{14} \circ \text{cut}_{14} = \text{id}$, the identity function on \mathcal{S} . The function cut_{14} maps every shuffle in \mathcal{S} to a distinct shuffle, and every shuffle in \mathcal{S} is the return value of cut_{14} for some shuffle in \mathcal{S} . This is what we call a **bijection**; see Section F.6.1.

See Section F.5.

For any possible value m of A_1 , we can in principle list all the shuffles among the $52!$ for which the event $\{A_1 = m\}$ occurs. If we apply cut_{14} to these shuffles, we get shuffles for which the event $\{A_4 = m\}$ occurs. In fact, we get all such shuffles because if s is a shuffle for which $\{A_4 = m\}$ occurs then $\text{cut}_{14}(s)$ is a shuffle for which $\{A_1 = m\}$ occurs. It follows the Kinds of A_1 and A_4 have the same weight on m , and since m was an arbitrary value, they have the same weight on all values, and hence the same Kind.

We can apply exactly the same argument with the function cut_{ij} for $i, j \in [1..5]$ with $i < j$. Hence, all the A_i 's have the same Kind. Moreover, using cut_{ij}

as a statistic swaps the values of A_i and A_j :

$$\begin{aligned}\text{proj}_{ij}(\text{between_aces}(S)) &= \langle A_i, A_j \rangle \\ \text{proj}_{ij}(\text{between_aces}(\text{cut}_{ij}(S))) &= \langle A_j, A_i \rangle.\end{aligned}$$

To illustrate this more concretely, we will study shuffles of the smaller “deck” with cards $1, 2, \dots, 5$ and a statistic analogous to `between_aces`, which we will call `two_four_gaps`, that uses $\{2, 4\}$ instead of the set \mathcal{A} .

```
pgd> T = shuffle(irange(5))
pgd> T
An FRP with value <2, 3, 5, 4, 1>. (It may be slow to evaluate its kind.)
```

With the statistic `two_four_gaps` defined in `frplib.examples.aces`, we have

```
pgd> G = two_four_gaps(T)
An FRP with values <0,2,1>
pgd> kind(G)
,---- 0.10000 ---- <0, 0, 3>
|---- 0.10000 ---- <0, 1, 2>
|---- 0.10000 ---- <0, 2, 1>
|---- 0.10000 ---- <0, 3, 0>
|---- 0.10000 ---- <1, 0, 2>
<> -|
|---- 0.10000 ---- <1, 1, 1>
|---- 0.10000 ---- <1, 2, 0>
|---- 0.10000 ---- <2, 0, 1>
|---- 0.10000 ---- <2, 1, 0>
`---- 0.10000 ---- <3, 0, 0>
pgd> kind(Sum(G))
<> ----- 1 ---- 3
pgd> kind(G[1])
,---- 0.4 ---- 0
|---- 0.3 ---- 1
<> -|
```



```

      |---- 0.2 ---- 2
      `---- 0.1 ---- 3
pgd> E(G[1])
1
pgd> Kind.equal(kind(G[1]), kind(G[2]))
True
pgd> Kind.equal(kind(G[1]), kind(G[3]))
True

```

So we see that G , the analogue of A , always has sum 3 and that the Kinds of its components G_1, G_2, G_3 are the same.

Let's check our argument with cut_{ij} .

```
pgd> kind(T)
```

I'm not showing the `kind(T)` tree here, but you should look at it. It's still small enough to understand and examine. Compare the Kind tree

```
pgd> kind(T) ^ cut(1,2, deck_size=5, aces={2, 4})
```

with `kind(T)`, where `cut` is a statistic factory defined in `frplib.examples.aces`. Observe that for each branch of `kind(T)` there is exactly one branch of `kind(T) ^ cut(1,2)` that swaps the first and second segment, and vice versa. If you examine these trees side by side and draw a line from each branch in `kind(T)` to the corresponding branch in `kind(T) ^ cut(1,2, deck_size=5, aces={2, 4})`, every branch in both trees will be accounted for. This is the analogue of the bijection we constructed in our earlier argument.

To try with these in the playground, do

```
pgd> from frplib.examples.aces import *
```

You might also find it interesting to look at the code to see how these statistics are defined.

This example shows us the power of the properties we have discovered about expectations. They let us compute the expectations without explicitly considering every possible value. This also shows that expectations are *often easier to compute than the full Kinds*.

Example 7.8. Let X be an FRP and let ψ, ϕ be two compatible statistics. Consider the conditional Kind that maps a value a to the Kind of $\phi(X) \mid \psi(X) = a$. In other words, we *observe* one transformed value of X and we want to use that information to *predict* a different transformed value of X . We will see how to compute the expectation of this Kind and in the process will discover a useful general property of expectations in the case where the two statistics are related.

As a concrete example, consider

```
pgd> Z = frp(either(0,1))
pgd> X = Z >> conditional_kind({
...>           0: uniform(1, 2, 3) * either(4, 5),
...>           1: either(7, 9, 1/7) * either(4, 5)
...>           })
pgd> X
An FRP of dimension 3 and size 10 with value <0, 3, 5>
pgd> kind(X)
,---- 1/12 ----- <0, 1, 4>
|---- 1/12 ----- <0, 1, 5>
|---- 1/12 ----- <0, 2, 4>
|---- 1/12 ----- <0, 2, 5>
|---- 1/12 ----- <0, 3, 4>
<> -|
|---- 1/12 ----- <0, 3, 5>
|---- 1/32 ----- <1, 7, 4>
|---- 1/32 ----- <1, 7, 5>
|---- 7/32 ----- <1, 9, 4>
`---- 7/32 ----- <1, 9, 5>
pgd> psi = Max
pgd> phi = Max - Min
```

Here, the statistic ψ computes the maximum component value, and the statistic φ computes the range of component values.

Suppose I want the risk-neutral price for $\text{phi}(X)$ having observed the fact that $\text{psi}(X) < 9$. We might want to write this as $E(\text{phi}(X) \mid (\text{psi} < 9))$ in the

playground, but as mentioned earlier, this does not work because the conditional constraint sees `phi(X)` but “forgets” `X`. We can apply `phi` after the conditional constraint with

```
E( phi(X | (psi < 9)) )
```

but the `@` operator gives us an equivalent expression that more closely tracks our mathematical notation $\mathbb{E}(\phi(X) \mid \psi(X) < 9)$:

```
pgd> E( phi @ X | (psi < 9) )
14/3
```

or if you like,

```
pgd> E( phi@X | (psi < 9))
14/3
```

Think of `phi @ X` (or equivalently `phi@X`), read “phi at X”, as evoking the idea of evaluating a function *at* an argument. The only difference from `phi(X)` is that it passes the value of `X` itself to the conditional constraint. This notation works with Kinds too, as we will see.

Now, we want to find $\mathbb{E}(\text{phi@X} \mid (\text{psi} == a))$ for each possible value `a` of `psi(X)`. For our concrete example this can be expressed as

```
E( (Max - Min) @ X | (Max == a) ).
```

(The parentheses around `Max == a` are necessary.) This gives our prediction of the range of `X`’s components given an observation of only the maximum component of `X`. Try evaluating these in the playground; you should get expectations 4, 5, 6, and 8 when `a` is 4, 5, 7, and 9, respectively.

Rather than just computing the expectations, it makes sense to consider the associated Kinds. In the playground, we can do

```
pgd> range_given_max = conditional_kind({
...>   4: phi @ kind(X) | (Max == 4),
...>   5: phi @ kind(X) | (Max == 5),
...>   7: phi @ kind(X) | (Max == 7),
```

```
...> 9: phi @ kind(X) | (Max == 9),
...> })
```

or with an “anonymous” function (denoted by `lambda` in Python):

```
pgd> range_given_max = conditional_kind(
...>     lambda a: phi @ kind(X) | (Max == a)
...> )
```

We’ll stick with the first form for now. Print this conditional Kind (in the first form) in the playground to see the results laid out nicely.

The `E` operator in the playground can compute all these risk-neutral prices as a package:

```
pgd> f = E(range_given_max)
```

which returns a *function* from `a` to the risk neutral price we want.

```
pgd> [f(a) for a in [4, 5, 7, 9]]
[4, 5, 6, 8]
```

Nice.

The previous example illustrates another useful property of expectations. The statistic φ in the example has a special form: it can be expressed as an operation on the value of ψ : $\varphi(x) = \zeta(x, \psi(x))$ for the function $\zeta(x, y) = y - \text{Min}(x)$. Our predictions about the value $\varphi(X)$ given the knowledge that $\psi(X) = a$ are thus the *same* as our predictions about $a - \text{Min}(X)$ given that same knowledge. We can check this in the playground:

```
pgd> E( (4 - Min)@X | (Max == 4) )
4
pgd> E( (Max - Min)@X | (Max == 4) )
4
```

Given that we know the value of $\psi(X)$, we can *substitute* that value in to the expression for $\phi(X)$ without changing our predictions.

This is one way to state the **substitution property** of expectations. It lets us use given information to *simplify* the expressions for the quantities we want to

predict.

Substitution Property. If X is an FRP and ψ, φ are compatible statistics where

$$\varphi(x) = \zeta(x, \psi(x))$$

for some functions ζ , then

$$\mathbb{E}(\zeta(X, \psi(X)) \mid \psi(X) = a) = \mathbb{E}(\zeta(X, a) \mid \psi(X) = a). \quad (7.26)$$

7.2 Computing Expectations

Let us take stock of what we have so far:

1. A precise definition of risk-neutral prices that captures our “best” prediction of a scalar FRP’s value and depends only on an FRP’s Kind.
2. A simple notation for the risk-neutral price of an FRP.
3. The idea of *expectation* that extends the risk-neutral price to FRPs of any dimension.
4. A set of key properties derived from the logic of the definition that expectations must follow for any FRP.

These are powerful enough already to compute predictions in many cases, but it would be nice if there were a direct way to express the expectation of an FRP/Kind. The good news is that there is and that we can find it from the properties of risk-neutral prices!

First, let’s consider an empirical to motivate the argument.

```
mkt> demo 10_000 with kind (<> 1 <-5> 4 <0> 3 <1> 2 <10>).
```

```
Activated 10000 FRPs with kind (<> 1 <-5> 4 <0> 3 <1> 2 <10>)
```

```
Summary of output values:
```

-5	1001 (10.01%)
0	4065 (40.65%)
1	3002 (30.02%)
10	1932 (19.32%)

```

mkt> demo 1_000_000 with kind (<> 1 <-5> 4 <0> 3 <1> 2 <10>).
Activated 10000 FRPs with kind (<> 1 <-5> 4 <0> 3 <1> 2 <10>)
Summary of output values:
-5      100466 (10.04%)
0       400263 (40.02%)
1       299594 (29.96%)
10      199677 (19.97%)

```

As we've seen earlier, the more FRPs we demo, the closer the relative frequencies in this table will get to the relative weights in the Kind. Now, suppose we purchase these FRPs for a price \$c for a large batch, then our payoff per unit will be approximately

$$-5 \cdot 0.1 + 0 \cdot 0.4 + 1 \cdot 0.3 + 10 \cdot 0.2 = 1.8,$$

where the approximation gets better and better as we purchases a larger and larger batch. If we choose a price $c < 1.8$, then for a sufficiently large batch, our payoff per unit will be positive. So any $c < 1.8$ is an arbitrage price. If $c > 1.8$, then for a sufficiently large batch, our payoff per unit will be negative, so no such price is an arbitrage price. And indeed, the risk-neutral price for this Kind is 1.8.

Even simpler, do the same calculation for the Kind of an event:

$$\langle \rangle \begin{cases} 1-p & \langle 0 \rangle \\ p & \langle 1 \rangle \end{cases}$$

In a large demo FRPs with this Kind, a proportion of roughly p of them will payoff \$1, with that proportion tending closer to p as the number of FRPs in the demo increases. If we pay more than \$ q per such FRP with $q > p$, then for any large enough demo with n FRPs, we are essentially certain to receive $< \$nq$. So q is an arbitrage price, and the risk-neutral price of this Kind is p .

If X is an FRP of size m with values v_1, v_2, \dots, v_m and respective weights w_1, w_2, \dots, w_m , we can define statistics

$$\psi_i(x) = \{x = v_i\}, \text{ for } i \in [1..m],$$

that equals 1 when its input equals v_i and 0 otherwise. Each transformed FRP $\psi_i(X)$

is an event – the event that X 's value equals v_i , and by the argument above

$$\mathbb{E}(\psi_i(X)) = \frac{w_i}{w_1 + w_2 + \cdots + w_m}. \quad (7.27)$$

Define the statistic

$$\xi(x) = v_1\psi_1(x) + v_2\psi_2(x) + \cdots + v_m\psi_m(x).$$

Notice that if we restrict attention to $x \in \{v_1, \dots, v_m\}$, then $\xi(x) = x$. Thus,

$$X = \xi(X) = v_1\psi_1(X) + v_2\psi_2(X) + \cdots + v_m\psi_m(X).$$

So, by the Scaling and Additivity properties and (7.27)

$$\begin{aligned} \mathbb{E}(X) &= \mathbb{E}(v_1\psi_1(X) + v_2\psi_2(X) + \cdots + v_m\psi_m(X)) \\ &= \mathbb{E}(v_1\psi_1(X)) + \mathbb{E}(v_2\psi_2(X)) + \cdots + \mathbb{E}(v_m\psi_m(X)) \\ &= v_1 \mathbb{E}(\psi_1(X)) + v_2 \mathbb{E}(\psi_2(X)) + \cdots + v_m \mathbb{E}(\psi_m(X)) \\ &= v_1 \frac{w_1}{w_1 + w_2 + \cdots + w_m} + \cdots + v_m \frac{w_m}{w_1 + w_2 + \cdots + w_m} \\ &= \frac{v_1w_1 + v_2w_2 + \cdots + v_mw_m}{w_1 + w_2 + \cdots + w_m}. \end{aligned} \quad (7.28)$$

Notice also that this argument holds *as is* with X of *any dimension*. The expectation of X is therefore a **weighted average of the FRP's possible values using the weights of each value from its Kind**. If the weights are from the canonical form of the Kind, then the denominator in the last equality would be 1.

This gives us a formula for the expectation of any FRP. The formula applies for FRPs of *any dimension* because we can scale and add tuples of common dimension.

Definition 22. If X is an FRP of any dimension with size m and values v_1, \dots, v_m and whose Kind has corresponding *canonical* weights p_1, \dots, p_m , then the expectation of X can be computed by

$$\mathbb{E}(X) = p_1v_1 + \cdots + p_mv_m. \quad (7.29)$$

If the Kind of X is in compact but not canonical form with corresponding weights

w_1, \dots, w_m , then equation 7.29 becomes

$$\mathbb{E}(X) = \frac{w_1 v_1 + \dots + w_d v_d}{w_1 + \dots + w_d}. \quad (7.30)$$

Expectations are thus *weighted averages* of the FRP's values.

We tend to use p 's to indicate canonical weights (that sum to 1) and w 's when this need not be true.

Example 7.9. If X has Kind described by $\langle \langle \rangle \ 1 \ \langle -1 \rangle \ 4 \ \langle 1 \rangle \rangle$,

$$\langle \rangle \begin{cases} \text{---} 1 \text{---} \langle -1 \rangle \\ \text{---} 4 \text{---} \langle 1 \rangle \end{cases}$$

what is $\mathbb{E}(X)$, $\mathbb{E}(X^3)$, and $\mathbb{E}(X \star X)$?

(Recall that X^2 and X^3 are “inlined” expressions for the transformed FRPs by statistics $\psi(x) = x^2$ and $\phi(x) = x^3$, respectively. See page 47.)

Applying equation 7.29 to each of these, we have

$$\begin{aligned} \mathbb{E}(X) &= \frac{1}{5} \cdot (-1) + \frac{4}{5} \cdot 1 = \frac{3}{5} \\ \mathbb{E}(X^3) &= \frac{1}{5} \cdot (-1) + \frac{4}{5} \cdot 1 = \frac{3}{5} \\ \mathbb{E}(X \star X) &= \frac{1}{25} \cdot \langle -1, -1 \rangle + \frac{4}{25} \cdot \langle -1, 1 \rangle \\ &\quad + \frac{4}{25} \cdot \langle 1, -1 \rangle + \frac{16}{25} \cdot \langle 1, 1 \rangle \\ &= \langle \frac{3}{5}, \frac{3}{5} \rangle. \end{aligned}$$

To see where the weights on the last two came from, compute the Kinds in the playground. You can enter `kind('(<> 1 <-1> 4 <1>'))` to get the Kind of X .

Example 7.10. I made two claims about the Kinds in Figure 44: (i) that the three Kinds have the same risk-neutral price, and (ii) that many people would *not* be indifferent between purchasing FRPs with these Kinds for \$10 because they differ in their *risk* of an adverse payoff. Let's confirm them.

Claim (i). The first Kind is just a constant with value 10, so its expectation is 10. The second Kind has values -1000 and 11 with weights 1 and 1010, so its

Part of the “Measuring Uncertainty” series.

expectation is

$$\frac{-1000 \cdot 1 + 11 \cdot 1010}{1011} = \frac{10110}{1011} = 10.$$

And the third Kind has values -10000 and 110 with weights 1 and 100.1, with expectation

$$\frac{-10000 \cdot 1 + 110 \cdot 100.1}{101.1} = \frac{1011}{101.1} = 10.$$

Claim (ii). To assess this, we will use the “variance shortcut” in equation (7.25) to compute the variance (a measure of uncertainty from Examples 7.6 and 7.4). If Z is an FRP with one of the three Kinds, then $\text{Var}(Z) = \mathbb{E}(Z^2) - (\mathbb{E}(Z))^2 = \mathbb{E}(Z^2) - 100$. For the first Kind, which is constant, there is no uncertainty, so the variance should be 0. And indeed: $10^2 - 100 = 0$. If Z has the second Kind,

$$\mathbb{E}(Z^2) = \frac{(-1000)^2 \cdot 1 + 11^2 \cdot 1010}{1011} = 1110,$$

so the variance is 1010. If Z has the third Kind,

$$\mathbb{E}(Z^2) = \frac{(-10000)^2 \cdot 1 + 110^2 \cdot 100.1}{101.1} = 1001100,$$

so the variance is 1001000. The uncertainty in the three Kinds increases as the possible values become more spread.

Example 7.11.

In the revisited disease testing example on page 225, we computed the Kind of an FRP whose outcome indicates whether someone has the disease when it is known that they test positive, $D \mid T = 1$

$$\langle \rangle \begin{cases} \text{999/1094} \text{ — } \langle 0 \rangle \\ \text{95/1094} \text{ — } \langle 1 \rangle \end{cases}$$

where the events D and T represents the disease status and test result, respectively. The FRP $D \mid T = 1$ is also an event and applying our formula, we have

$$\mathbb{E}(D \mid T = 1) = \frac{999}{1094} \cdot 0 + \frac{95}{1094} \cdot 1 = \frac{95}{1094} \approx 0.0868.$$

This tells us that after observing a positive test result, our prediction is that the patient has only about an 8.6% chance of having the disease.

This illustrates another common pattern. An event⁷³ acts as an indicator of whether something happens. Equation (7.29) shows that for any event I , $\mathbb{E}(I)$ is just the canonical weight on its 1 branch.⁷⁴ We interpret this quantity as a measure of how likely that event is to occur. In that sense, the disease-testing example is suprising in that the disease remains unlikely after a positive test.

⁷³Recall: FRP whose only values are 0 and 1.

⁷⁴The discussion just after Puzzle 54 also made good use of this fact.

Example 7.12. Try these calculations; look at the weighted averages that you get; and compare them to the expectations that we compute. I’ve omitted the results here.

```
pgd> coin = either(0, 1) # Model: 1 for heads, 0 for tails; equally weighted
pgd> flips10 = coin ** 10
pgd> num_heads_in_10 = Sum(flips10)
pgd> num_heads_in_10
pgd> E(num_heads_in_10)
pgd> E(num_heads_in_10 - 5) # We know this from Additivity and Constancy. Why?
```

We can see that the Kind’s canonical form shows us the weighs and values and can confirm that the resulting weighted average is just the risk-neutral price. Notice how the Properties we discovered earlier help us compute the last value without actually hitting enter.

For a value like $E(\text{num_heads_in_10})$, there are two ways to think about the computation. We can look at the Kind `flips10` and average up the sum of components for each value in that Kind with the given weights. Or, we can generate the *new* Kind `num_heads_in_10` and take a direct weighted average. *Both ways give the same answer.*

Let’s consider that in a smaller case. Look at both Kinds and do the calculation from each tree:

```
pgd> coin ** 4
pgd> Sum(coin ** 4)
pgd> E(Sum(coin ** 4))
```

The reason these give the same answer is that we defined the transformed Kind by passing the values through the statistic and combining equal weights.

Example 7.13. Continuing the previous example, let us ask at which of 16 flips of our coin does the first heads occur.

```
pgd> flips16 = coin ** 16
```

We define a statistic that answers our question

```
pgd> @scalar_statistic(description='Index of first heads, or 1000')
...> def first_heads(x):
...>     return 1 + index_of(1, x, not_found=999)
```

This has type $n \rightarrow 1$ for every $n \in [1..1000]$, and returns 1000 (arbitrarily) if a heads did not occur.

```
pgd> when_heads = first_heads(flips16)
```

Look at the following Kinds:

```
pgd> when_heads
pgd> when_heads ~ IfThenElse(__ > 10, 1000, __)
pgd> (when_heads | (__ > 4)) ~ IfThenElse(__ == 1000, __, __ - 5)
```

What do each of these Kinds mean? How do they compare? Can you explain?

The first is the Kind of the number of flips to the first heads in 16 flips. The second is a transform of the Kind `when_heads`, where we map every value bigger than 10 to the arbitrary value 1000. We do this to ease comparison with the third Kind. The third is the Kind of an FRP that gives the number of 1's (heads) *after* having observed that there are no 1's (heads) in the first four flips. The statistic at the end shifts the values back to the scale of the first two Kinds.

Example 7.14 Entropy

Example 7.4 introduced the *entropy* of an FRP/Kind as a measure of uncertainty. Unlike variance or mean absolute deviation, the entropy does not predict the

Part of the “Measuring Uncertainty” series.

distance between value and risk-neutral price but rather predicts the size of the weight on the FRP's value. Here, we will use equation (7.29) to get an expression for the entropy and develop a first interpretation of its meaning.

Let Z be an FRP with Kind K in canonical form. If z is a possible value of Z , we write $K(z)$ for the weight associated with branch z , treating K as a function. Then, applying (7.29), we have

$$\begin{aligned}\mathbb{H}(Z) &= \mathbb{E}(-\lg K(Z)) \\ &= \sum_{z \in \text{values}(Z)} K(z)(-\lg K(z)) \\ &= - \sum_{z \in \text{values}(Z)} K(z) \lg K(z).\end{aligned}\tag{7.31}$$

The terms in the sum are well defined for any value of the weights: the function $\langle p \rangle \mapsto -p \lg p$ goes to 0 as p goes to 0 because $-\lg p$ grows much more slowly than p shrinks. We can thus write $\mathbb{H}(Z) = -\sum_z K(z) \lg K(z)$, taking $K(z) = 0$ for any z that is not a possible value of Z .

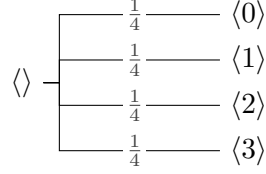
To start, assume that we have a large supply of balanced, independent coin flips, i.e., events $C_{[1]}, C_{[2]}, C_{[3]}, \dots$ with Kind F $\langle \rangle \begin{array}{c} \text{---} \frac{1}{2} \text{---} \langle 0 \rangle \\ \text{---} \frac{1}{2} \text{---} \langle 1 \rangle \end{array}$, where 0 represents tails and 1 represents heads. Think of a coin flip as a random bit, so a single flip represents 1 bit of randomness.

Suppose we have another FRP X with Kind F . This has entropy

$$\mathbb{H}(X) = -\frac{1}{2} \lg \frac{1}{2} - \frac{1}{2} \lg \frac{1}{2} = \frac{1}{2} + \frac{1}{2} = 1.$$

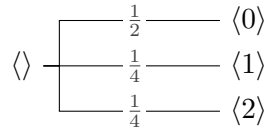
Entropy is measured in *bits*, and this result tells us that X has 1 bit of “uncertainty.” One way to interpret this is that it takes 1 coin flip to simulate a clone of X . For instance, we can take the value of $C_{[1]}$ as the value of our clone. This has the same Kind as X .

Let Y be an FRP with Kind



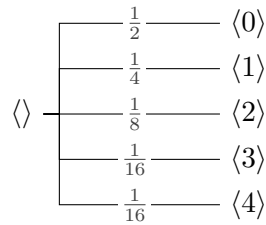
Then, $\mathbb{H}(Y) = -\frac{1}{4} \cdot (-2) - \frac{1}{4} \cdot (-2) - \frac{1}{4} \cdot (-2) - \frac{1}{4} \cdot (-2) = 2$. We can simulate a value of $\text{clone}(Y)$ with two coin flips, say $C_{[2]}$ and $C_{[3]}$, with the former's value determining the ones-bit of the simulated value and the latter's value determining the twos-bit of the simulated value: $2C_{[3]} + C_{[2]}$. Two coin flips, two bits of uncertainty.

The FRP Z with Kind



has entropy $\mathbb{H}(Z) = -\frac{1}{2} \cdot (-1) - \frac{1}{4} \cdot (-2) - \frac{1}{4} \cdot (-2) = 1.5$ bits. So this Kind embodies 1.5 bits of uncertainty.

But what does it mean to require 1.5 coin flips? Well, to simulate a value of $\text{clone}(Z)$, we can start with a coin flip $C_{[4]}$. *If* its value is 0 (tails), we use 0 as the simulated value and we are done, which requires 1 flip. If $C_{[4]}$'s value is 1, however, we pull out $C_{[5]}$ and give a simulated value of 1 or 2 as this flip has value 0 (tails) or 1 (heads), which requires 2 flips. So half the time we need only 1 flip and half the time 2 flips – thus 1.5 flips.



The entropy of an FRP with the Kind above is 2.375. Explain how this value is derived. How would you use coin flips $C_{[6]}, C_{[7]}, C_{[8]}, C_{[9]}$ to simulate the value

of an FRP with this Kind? How might you interpret the value of the entropy in light of this?

You need not use all four coin flips for every value.

The `entropy` function in the playground computes the entropy of a Kind or FRP. For instance, try

```
pgd> entropy(uniform(1, 2, ..., 16))
```

One interpretation of entropy is that it measures how many bits of randomness are used on average in generating the value of an FRP. We will see other, related interpretations later.

7.3 Kinds and Expectations

Because the FRP's expectation depends only on the FRP's Kind, we can therefore talk about the expectation of an FRP or the risk-neutral price of a Kind, as convenient. The expectation represents a *prediction* of an FRPs value, a notion of “typical value” that is *in some sense* as close as possible to whatever value is produced by the FRP. In some sense?? What does that mean?

Let X be a scalar FRP and consider a family of statistics $\psi_c(x) = (x - c)^2$ for every real number c . Define a function

$$L(c) = \mathbb{E}(\psi_c(X)) = \mathbb{E}((X - c)^2),$$

where the second form is the inline expression of the statistic.⁷⁵ This function gives the risk-neutral price of an FRP that represents *the squared distance between the value of X and the number c* . Thus, $L(c)$ measures for each c how “close” c is on average to the value of X . The value that *minimizes* $L(c)$ is $\mathbb{E}(X)$. To see this apply the properties of expectations:

⁷⁵See page 47.

$$\begin{aligned} L(c) &= \mathbb{E}((X - c)^2) \\ &= \mathbb{E}((X - \mathbb{E}(X) + \mathbb{E}(X) - c)^2) && \textcircled{1} \\ &= \mathbb{E}((X - \mathbb{E}(X))^2 + 2(\mathbb{E}(X) - c)(X - \mathbb{E}(X)) + (\mathbb{E}(X) - c)^2) && \textcircled{2} \\ &= \mathbb{E}((X - \mathbb{E}(X))^2) + \mathbb{E}(2(\mathbb{E}(X) - c)(X - \mathbb{E}(X))) + \mathbb{E}((\mathbb{E}(X) - c)^2) && \textcircled{3} \\ &= \mathbb{E}((X - \mathbb{E}(X))^2) + 2(\mathbb{E}(X) - c)\mathbb{E}(X - \mathbb{E}(X)) + (\mathbb{E}(X) - c)^2 && \textcircled{4} \\ &= \mathbb{E}((X - \mathbb{E}(X))^2) + 2(\mathbb{E}(X) - c)0 + (\mathbb{E}(X) - c)^2 && \textcircled{5} \end{aligned}$$

$$= \mathbb{E}((X - \mathbb{E}(X))^2) + (\mathbb{E}(X) - c)^2.$$

In ①, we add $0 = \mathbb{E}(X) - \mathbb{E}(X)$ inside quadratic. In ②, expand the quadratic in terms of $X - \mathbb{E}(X)$ and $\mathbb{E}(X) - c$. ③ applies Additivity. ④ applies Scaling on the middle term with constant $2(\mathbb{E}(X) - c)$ and Constancy on the last term. In ⑤, the middle term cancels by Constancy, Scaling, and Additivity because

$$\mathbb{E}(X - \mathbb{E}(X)) = \mathbb{E}(X) - \mathbb{E}(\mathbb{E}(X)) = \mathbb{E}(X) - \mathbb{E}(X) = 0.$$

Finally, we are left with two terms: the first *does not depend on c* and the second is a simple quadratic in c minimized at $c = \mathbb{E}(X)$. So, $c = \mathbb{E}(X)$ minimizes $L(c)$.

If we define $|x| = \sqrt{x_1^2 + x_2^2 + \cdots + x_d^2}$ for tuples x of dimension d . This argument carries over, almost directly, to FRPs of any dimension. Here, we use a family of statistics $(|\blacksquare - c|^2)$, where c is a tuple of dimension d and $x - c$ is the component-wise difference. This reduces to ψ_c in the scalar case.

If X is an FRP of dimension d , its expectation $\mathbb{E}(X)$ is a tuple of dimension d that minimizes the *predicted squared prediction error*

$$\langle c \rangle \mapsto \mathbb{E}(|X - c|^2). \quad (7.32)$$

over all d -tuples c .

When X is a scalar FRP, the minimum value $\mathbb{E}(|X - \mathbb{E}(X)|^2)$ is called the **variance** of X , denoted $\text{Var}(X)$, as discussed in Examples 7.4 and 7.6. (The idea of variance also extends to the multi-dimensional case but with some nuance, so we will discuss it later.)

Thus, the sense in which the expectation is an optimal prediction of an FRP's value is it that the expectation minimizes our “mean squared” prediction error. The expectation is a “typical value” that is on average as close as possible to the value produced by the FRP. As we proceed, we will see several different interpretations of expectations, and we will emphasize the connection to risk-neutral prices.

Puzzle 52. Consider a Kind like `uniform(1, 2, ..., 999999)`. For each $j \in [1..999999]$, what is $\mathbb{E}(|X - j|^2)$, where FRP X has this Kind? (You can do this by reasoning or by computation.) Which value gives the smallest predicted prediction error?

We have seen that the expectation of an FRP is determined by the FRP’s Kind. It turns out the Kind of an FRP is determined by expectations of *all transforms* of the FRP by compatible statistics.

Property 23. Two FRPs X and Y have the same Kind if and only if they have the same set of possible values and

$$\mathbb{E}(\psi(X)) = \mathbb{E}(\psi(Y)) \quad (7.33)$$

for *every compatible statistic* ψ .

This tells us that, *in aggregate*, the expectations of transformed FRPs/Kinds contain all the information needed to determine the Kind of the original FRP. The word “determine” here does not necessarily mean that we can compute the Kind directly from this information (though we often can) but rather that any computation, analysis, or prediction that depends only on the Kind will be the same for any FRP with that Kind. This needs a little unpacking.

In one direction, Property 23 says that two FRPs with the same Kind yield equal expectations when transformed by the same statistic. The other direction *seems* less useful because it requires that the expectations of transformed FRPs be the same for all compatible statistics. (It’s hard to compute the expectation in practice for *all* compatible statistics.) Fortunately, this direction also holds with smaller collections of statistics as long as the collection is sufficiently rich. In other words, we can “determine” the Kind of an FRP with the expectations of a well-chosen collection of statistics.

Definition 24. A collection of statistics \mathcal{S} is said to **determine** the Kind of compatible FRPs if for any FRPs X, Y that are compatible with all the statistics

in \mathcal{S}

$$\mathbb{E}(\psi(X)) = \mathbb{E}(\psi(Y)) \text{ for all } \psi \in \mathcal{S} \text{ implies } \text{kind}(Y) = \text{kind}(X). \quad (7.34)$$

That is, if X and Y have the same expectation when transformed by all statistics in \mathcal{S} , they have the same risk-neutral price when transformed by *all* compatible statistics.

Section 2 emphasized that statistics often represent questions whose answers we would like to predict. It is useful then to view a collection of statistics as a set of questions we might want to answer. These determine the Kind of an FRP if the predicted answer to these questions uniquely specify the predicted answers to all questions we might ask.

This requirement is not trivial. For $\mathcal{S} = \{\text{id}\}$, the singleton collection containing only the identity function, this would require that $\mathbb{E}(X) = \mathbb{E}(Y)$ implied that X and Y had the same Kind, but that is not true, as we saw for instance in Figure 44.

Puzzle 53. For $\mathcal{S} = \{\text{id}, (\blacksquare^2)\}$, find two distinct Kinds that give the same expectation for all statistics in \mathcal{S} .

As a positive example, let \mathcal{E} be the collection of statistics of type $1 \rightarrow 1$ that indicate a specific value:

$$\mathcal{E} = \{ \{ \blacksquare = c \} \mid c \text{ is a real number} \}.$$

(Recall that the indicator $\{ \blacksquare = c \}$ is the function that returns 1 when its argument equals c and 0 otherwise. See F.4 and F.3.) For a scalar FRP X , the transformed FRPs by the statistics in \mathcal{E} are all the events $\{X = c\}$ for real number c . If c is not a possible value of X , then $\mathbb{E}(\{X = c\}) = 0$. If c is a possible value of X , then $\mathbb{E}(\{X = c\})$ is the canonical weight on value $\langle c \rangle$ in $\text{kind}(X)$. Another scalar FRP Y with $\mathbb{E}(\{Y = c\}) = \mathbb{E}(\{X = c\})$ for all c must have the same possible values and the same weights – and so the same Kind. Thus, \mathcal{E} determines the Kind of any scalar FRP.

Another example is the collection of indicator statistics \mathcal{F} of type $1 \rightarrow 1$

$$\mathcal{F} = \{ \{ \blacksquare \leq c \} \mid c \text{ is a real number} \}.$$

For a scalar FRP X , $\mathbb{E}(\{X \leq c\})$ adds up the weight for all values of X that are at most c . So by varying c , we can find X 's value (where the expectation jumps) and the weight at that value (the size of the jump). So if $\mathbb{E}(\{X \leq c\}) = \mathbb{E}(\{Y \leq c\})$ for all c , X and Y must have the same Kind, and \mathcal{F} determines the Kind of any scalar FRP.

Let \mathcal{T} be the set of three statistics $(\frac{1}{2}(\blacksquare - 1)(\blacksquare - 2))$, $(-\frac{1}{2}\blacksquare(\blacksquare - 2))$, and $(\frac{1}{2}\blacksquare(\blacksquare - 1))$ restricted to the domain $\{0, 1, 2\}$. For any FRP X with possible values $\{0, 1, 2\}$, the expectations $\mathbb{E}(\psi(X))$ for $\psi \in \mathcal{T}$ give the canonical weights on 0, 1, and 2. The statistics in \mathcal{T} thus determine the Kind of any FRPs with only those three possible values.

Puzzle 54. If Z is an FRP of dimension 2, the collection $\text{proj}_1, \text{proj}_2$ does *not* determine the Kind of Z . Find an example to demonstrate this.

Specifically, you need another FRP U with the same possible values as Z where $\mathbb{E}(\text{proj}_1(Z)) = \mathbb{E}(\text{proj}_1(U))$ and $\mathbb{E}(\text{proj}_2(Z)) = \mathbb{E}(\text{proj}_2(U))$ but Z and U have *different Kinds*.

Looking back to Figure 13, a useful and powerful perspective on how we use probability theory in practice is that we measure some data from whatever system we are studying and try to predict the answers to a range of questions about those data. For each question, we want a predicted answer.

If X is an FRP, like the Data FRP in the Figure, we can define an operator D_X that does exactly that: maps questions to predicted answers.

$$D_X(\psi) = \mathbb{E}(\psi(X)). \quad (7.35)$$

This takes as input a statistic that is compatible with X – that is, a question we can ask about our data – and returns as output a predicted answer to that question.

Property 23 tells us that the expectations $\mathbb{E}(\psi(X))$, varying over compatible statistics ψ , determine the Kind of X . So D_X embodies the same information as $\text{kind}(X)$ in a different form. We can say that our *knowledge* about some random quantity is reflected by how well we can make predictions about the quantity. Both $\text{kind}(X)$ and D_X thus fully describe what we know about X 's value. As the system evolves, we may learn more information about X 's value (expressed through conditional constraints), and our knowledge changes (to $\text{kind}(X \mid C)$ and $D_{X|C}$ for

conditional constraint C).

The D operator is available in the playground as D_* . It takes an FRP or Kind as input and returns a *function* mapping every (compatible) statistic to its corresponding prediction. If X is an FRP, $D_*(X)$ is a function that acts on statistics ψ to give

$$D_*(X)(\psi) = E(\psi(X)).$$

For example,

```
pgd> X = frp(uniform(1, 2, 3) * uniform(1, 2, 3))
pgd> f = D_*(X)
pgd> f(Sum)      # == E(Sum(X))
4
pgd> f(Max)      # == E(Max(X))
22/9
pgd> f(Min)      # == E(Min(X))
14/9
```

7.4 Probabilities are the Expectations of Events

In practice, we are often interested in whether particular events occur. Because an event is an FRP with possible values 0 and 1, the Ordering property (7.11) tells us that the expectation of an event is a *number between 0 and 1*. This is a prediction of how likely the event is to occur, increasing from an essential certainty that it will not occur at 0 to an essential certainty that it will occur at 1. with value 0 meaning that it will.

Thus for any event V , the expectation $E(V)$ satisfies $0 \leq E(V) \leq 1$ and measures in some sense our confidence that the event V will occur. Because we use such expectations so often, we give them a distinctive name.

Definition 25. A **probability** is the expectation of an event. It is a number in $[0, 1]$ that measures our prediction of whether the event will occur.

If V is an event, $E(V)$ is called the *probability of V* . Events with higher probability are said to be more *likely* than events with lower probability.

Probabilities are nothing new. They are just expectations – risk-neutral prices –

for FRPs that can have particular values. They inherit all the properties of general expectations and are computed according to the same rules.

If X is an FRP and if v is a possible value of X , then $\{X = v\}$ is the event that occurs when X produces value v and does not occur if it produces another value. The Kind of $\{X = v\}$ looks like

$$\langle \rangle \begin{cases} 1-p & \langle 0 \rangle \\ p & \langle 1 \rangle \end{cases}$$

where p is the canonical weight on v in $\text{kind}(X)$. Applying equation (7.29), we see that $\mathbb{E}(\{X = v\}) = p$. **The canonical weight on value v in the Kind $\text{kind}(X)$ is the probability that X has value v .**

Two events V and W are *complements* if $V + W = 1$. This means that exactly one of the two events must occur. If V occurs, then we know W does not, and vice versa. Applying Additivity ($\mathbb{E}(V + W) = \mathbb{E}(V) + \mathbb{E}(W)$) and Constancy ($\mathbb{E}(1) = 1$), we can take expectations on both sides of this equation to find

$$\mathbb{E}(V) + \mathbb{E}(W) = 1.$$

Complementary events have probabilities that sum to 1.

We frequently specify events with Boolean expressions in *Iverson braces*,⁷⁶ like $\{X = 4\}$, $\{Y > 10\}$, and $\{U = 4 \wedge 0 \leq V \leq 10\}$. When taking the expectations, we typically put the FRP in parentheses after the \mathbb{E} , like $\mathbb{E}(\{X = 4\})$, $\mathbb{E}(\{Y > 10\})$, and $\mathbb{E}(\{U = 4 \wedge 0 \leq V \leq 10\})$. However, in this specific case where the FRP is a single event in Iverson braces, the extra parentheses may seem superfluous, and we treat them as optional. So $\mathbb{E}\{X = 4\}$ and $\mathbb{E}(\{X = 4\})$ mean the same thing.

⁷⁶See page 204.

Notational Option. When taking the expectation of a *single event in Iverson braces*, the parentheses around the argument to \mathbb{E} are optional.

Note that this convention does not apply to FRPs that are expressed as multiple terms, such as $X \cdot \{X > 2\}$ or $\{Y = 2\} + \{Z < 4\}$, we keep the parentheses.

Checkpoints

After reading this section you should be able to:

- Define an arbitrage price and the risk-neutral price for a scalar FRP.
- Use the market to estimate risk-neutral prices for simple FRPs.
- Use the definitions to guess at some basic properties of risk-neutral prices.
- Define the expectation of an n -dimensional FRP / Kind for $n \geq 1$.
- Explain the Constancy, Scaling, Ordering, and Additivity properties of expectations.
- Describe various measures of uncertainty.
- Explain the formula for expectations from a Kind.
- Describe briefly why the expectation depends only on the Kind and not on the FRPs value.
- Describe how the expectations of transformed FRPs determine a Kind.
- Define probabilities in terms of expectations.

8 Patterns, Predictions, and Practice

Key Take Aways

In this section, we solve a variety of examples that shows the power of the tools we have developed. Throughout these examples, we see that four *basic operations* are combined to produce all the calculations we need. These are

- Transforming with Statistics
- Building with Mixtures
- Constraining with Conditionals
- Predicting with Expectations

Several patterns in the use of these four operations arise frequently: marginalization (projecting onto selected components), *conditioning* (the method of hypotheticals), Bayes's Rule (reversing the conditionals), and solving various iterative and recursive equations.

As we move forward to develop the mathematical parts of the theory, it will be helpful to see how these four rather mundane operations underlie all of our analysis, even in the more abstract settings that deal with infinities and infinitesimals.

In this section, we will use the `frplib` playground to tackle a wide variety of example problems that synthesize the ideas of the previous sections. These problems illustrate both essential techniques of probability theory and common patterns of analysis. Focus here on identifying the Big 3+1 operations – transformations, mixtures, conditionals, and expectation – and studying how they are combined to solve problems. All of the complex analyses we do and will do are built from these basic operations, and the commentary on the examples will attempt to highlight this.

The section is loosely divided into subsections emphasizing particular techniques. In the example playground code for this section, the prompts (like `pgd>` and `...>`) are sometimes omitted from the display to save space unless there is output to show. You can load each example's code into the playground using the name in lower case with no punctuation and `_` instead of space, but only up through the first three words,

e.g., “`from frplib.examples.six_of_one import *`”. You are encouraged to follow along with the examples in the playground as you read.

By understanding how the big 3+1 operate – transforming values, erasing branches, combining stages, taking weighted averages – we can learn to recognize and exploit these operations even in more complicated calculations and contexts.

8.1 Types and Operations

The examples in the following subsections make heavy use of the `frplib` playground to show both how we *model* the situation at hand and how we use the Big 3+1 to understand and analyze that model. To make that easier, it is helpful to keep in mind the *types* of the various entities we are working with. A type is a set of objects with a set of basic operations on those objects. Examples include the *primitive types* that we use regularly: natural numbers $\mathbb{N} = [0..)$, integers \mathbb{Z} , real numbers \mathbb{R} , Booleans $\mathbb{B} = \{\perp, \top\}$, and the *unit type* $\mathbb{U} = \{\langle \rangle\}$, which contains a single object.⁷⁷

⁷⁷See [F.1](#) for more on all these sets.

There are four *basic types* in the playground:

- A **Quantity** is a number or a symbolic value representing a number.
- A **Value** is a tuple whose components are of type **Quantity**. We elide the distinction between a tuple of dimension 1 and the scalar quantity it contains. Booleans are represented as scalars with 0 for false (\perp) and 1 for true (\top).
- A **Kind** is a tree in canonical form with weights that are positive quantities and leaf nodes of type **Value**, where all leaves are distinct and have the same dimension.
- An **FRP** is a device that produces a single **Value** when activated that is fixed for all time.

For the last three types, we add a subscript (like **Value_d**) to restrict the type to objects of dimension d . The type **Value₀** is just another name for the unit type \mathbb{U} .

We specify *function types* as $a \rightarrow b$ as the set of functions that take input of type a and return output of type b . We write $f: a \rightarrow b$ to indicate that function f has type $a \rightarrow b$. A function $c: \mathbb{U} \rightarrow b$ from the unit type to some other type b is for all practical purposes equivalent to the object $c(\langle \rangle)$ of type b , and we treat it as such.

For a binary operator op , we write $a \text{ op } b \rightarrow c$ to indicate that the operator takes data of type a on the left and type b on the right and produces a result of type c .

We define three primary function types in the playground. For instance, a statistic is a function that takes and returns a **Value**. Broadly speaking these are:

```
Statistic: Value → Value
ConditionalKind: Value → Kind
ConditionalFRP: Value → FRP
```

but more precisely, we specify for each its codimension c and dimension d :

```
Statisticc,d: Valuec → Valued
ConditionalKindc,d: Valuec → Kindd
ConditionalFRPc,d: Valuec → FRPd
```

Objects of all three types may in practice be defined on strict subsets of **Value_c**. **Condition** is a sub-type of **Statistic** of statistics returning Boolean values.

Recall our observation that a **Kind** is just a conditional Kind of codimension 0 and similarly an **FRP** is just a conditional FRP of codimension 0. This relates to the comment earlier that a function from the unit type **Value₀** to some other type b is equivalent to an object of type b . So we treat **Kind_d** as equivalent to the type **ConditionalKind_{0,d}** and **FRP_d** as equivalent to the type **ConditionalFRP_{0,d}**.

We have seen several combinators that can operate on Kinds or FRPs:

- $\text{FRP}_m \wedge \text{Statistic}_{m,n} \rightarrow \text{FRP}_n$
 $\text{Kind}_m \wedge \text{Statistic}_{m,n} \rightarrow \text{Kind}_n$
- $\text{Statistic}_{m,n}(\text{FRP}_m) \rightarrow \text{FRP}_n$
 $\text{Statistic}_{m,n}(\text{Kind}_m) \rightarrow \text{Kind}_n$
- $\text{FRP}_m \star \text{FRP}_n \rightarrow \text{FRP}_{mn}$
 $\text{Kind}_m \star \text{Kind}_n \rightarrow \text{Kind}_{mn}$
- $\text{FRP}_m \star \star \text{NaturalNumber}_n \rightarrow \text{FRP}_{m^n}$
 $\text{Kind}_m \star \star \text{NaturalNumber}_n \rightarrow \text{Kind}_{m^n}$
- $\text{ConditionalFRP}_{m,n} \triangleright \text{ConditionalFRP}_{n,r} \rightarrow \text{ConditionalFRP}_{m,r}$
 $\text{ConditionalKind}_{m,n} \triangleright \text{ConditionalKind}_{n,r} \rightarrow \text{ConditionalKind}_{m,r}$

- $\text{FRP}_m \mid \text{Condition}_m \rightarrow \text{FRP}_m$
 $\text{Kind}_m \mid \text{Condition}_m \rightarrow \text{Kind}_m$
- $\text{ConditionalFRP}_{m,n} \parallel \text{FRP}_m \rightarrow \text{FRP}_n$
 $\text{ConditionalKind}_{m,n} \parallel \text{Kind}_m \rightarrow \text{Kind}_n$
- $\text{Statistic}_{m,n} @ \text{FRP}_m \rightarrow \text{FRP}_n$
 $\text{Statistic}_{m,n} @ \text{Kind}_m \rightarrow \text{Kind}_n$

Although we write `stat(k)` or `stat(X)` for a statistic `stat` and a Kind `k` or FRP `X`, we think of this as an *operator* (equivalent to \sim), *not* as evaluating the statistic with an argument. A statistic does **not** take a Kind or FRP as input, only a `Value`.

Finally, the playground allows combining statistics with simple expressions to produce new statistics.

```
Statistic + Statistic → Statistic
Statistic + Value → Statistic
...
```

and similarly with other arithmetic and comparison operators
(e.g., `-`, `*`, `/`, `**`, `%`, `==`, `<=`).

Keeping this firmly in mind, we are ready to dive in to some examples.

8.2 Simple Finite Random Processes

In this subsection, we look at random processes with a fixed, finite number of stages. Sometimes these stages interact, sometimes they do not. As you consider how to model these examples, look for the parts of the process that are easier to understand in isolation. If those parts require some knowledge of earlier stages, that's OK – it's why we have mixtures. That suggests defining a conditional Kind/FRP. If we can model a part of the process independently of any other stage, then we can combine it with other parts with an independent mixture. Throughout, we think hard about how to represent the quantities we are modeling. This sometimes involves assigning meaning to arbitrary numbers, and it sometimes requires us to think about how we describe the state of the system. Statistics are useful for building an initial state and for transforming between different representations.

Example 8.1 Doubled Cards

A deck of 100 cards is labeled $1, 2, \dots, 100$. You choose two cards from the deck in succession. The deck is well shuffled, so you can assume that every pair of cards has equal weight to be selected.

Define two FRPs:

- D is the event that the second card's number is exactly twice the first card's.
- T is the event that *either* card's number is twice the other.

Find the expectations $\mathbb{E}(D)$ and $\mathbb{E}(T)$.

First, because D and T are both events, their expectations are the probabilities that the events occur. If D occurs, then by definition, T occurs as well, so the value produced by D is always \leq the value produced by T . We write this as $D \leq T$, and by the ordering property $\mathbb{E}(D) \leq \mathbb{E}(T)$. The event T is more likely to occur.

Second, we can recognize a mixture structure here: we first draw a card from the deck and then draw a second card from the deck that is missing the first card. At both stages, all cards remaining in the deck are selected with equal weight. We define `draw` as the Kind of the mixture FRP that represents the pair of card numbers drawn from the deck:

```
first_card = uniform(1, 2, ..., 100)
all_cards = first_card.values
second_card = conditional_kind({
    first: uniform(all_cards - {first}) for first in all_cards
})
draw = first_card >> second_card
P = frp(draw)
```

Here, `first_card.values` is the *set* of possible values for the first card and `{first}` is the singleton set with just the “current” card; the difference removes the latter from the former. The Kind `draw` has dimension 2 and size 9900; its values are pairs of distinct card numbers. Because `first_card` and each Kind in `second_card` are uniform, the mixture Kind has the same weight, $1/9900$, on every branch. You can see this by looking at `draw` in the playground, though

the large size makes that less than convenient.

P is the FRP representing the drawn pair of cards, and D and T are transforms P by two statistics:

```
is_card_doubled = Proj[2] == 2 * Proj[1]
is_either_doubled = Or(is_card_doubled, Proj[1] == 2 * Proj[2])
```

```
D = is_card_doubled(P)
T = is_either_doubled(P)
```

The first just checks if the second component of a value is equal to twice the first component, and the second checks that and in the reverse direction.

The Kinds D and T are transforms of **draw** by these statistics:

```
D_kind = is_card_doubled(draw)      # same as kind(D)
T_kind = is_either_doubled(draw)    # same as kind(T)
```

Think about how we find $\text{kind}(D)$ by transforming **draw**. Values with doubled cards map to $\langle 1 \rangle$; the rest map to $\langle 0 \rangle$. The weights in each branch of $\text{kind}(D)$ add up the weights for all branches of **draw** with the corresponding value. Because all the weights of **draw** equal $1/9900$, the weights are $(9900 - b_1)/9900$ and $b_1/9900$ where b_1 is the number of **draw**'s branches for which **is_card_doubled** equals $\langle 1 \rangle$. As D is an event, $\mathbb{E}(D)$ equals the weight on $\langle 1 \rangle$ in $\text{kind}(D)$. A similar approach works for T . Looking at the results:

```
pgd> D_kind
      ,---- 0.99495  ----- 0
<> -|
      `---- 0.0050505  ---- 1
pgd> E(D_kind)      # == E(D)
1/198
pgd> E(T_kind)      # == E(T)
1/99
pgd> FRP.sample(10_000, D)
+-----+-----+-----+-----+
```

Values	Count	Proportion
=====+=====+=====		
0	9946	99.46%
1	54	0.54%
+-----+-----+-----		

So, for 50 of the possible pairs, `is_card_doubled` is true and for 100 of the pairs, `is_either_doubled` is true. We can reason that if the first card is bigger than 50, the second card cannot double it, and if it is in $[1..50]$ only 1 out of the 99 remaining cards that will make D occur. T occurs for these 50 possibilities and also for the 50 pairs the second card is in $[1..50]$ and the first card doubles the first. The probabilities of these events are thus $1/198$ and $1/99$, and the demo of FRPs with Kind matching D is close to that proportion.

Our questions are answered, but it is worth looking at a few playground variations. First, we defined the conditional Kind `second_card` using a dictionary, but we could also use a function:

```
second_card = conditional_kind(
    lambda first: uniform(all_cards - {first}),
    codim=1
)
```

The `lambda first` defines a Python anonymous function that takes a single argument `first`, a value of `first_card`. The `codim=1` argument to `conditional_kind` tells it to expect a function of a *scalar* argument, so `first` can be a number.

Second, applying our reasoning *in advance*, we could define the conditional Kind `second_card` by

```
conditional_kind(
    lambda first: either(0, 1, 98) if c <= 50 else constant(0),
    codim=1
)
```

Finally, there is a built-in factory in the playground that creates `draw` directly

```
ordered_samples(2, irange(1, 100))
```

This example illustrates a common pattern: the transformed mixture. We describe a system in stages, because the stages are usually simpler to describe and specify. We use mixtures to combine the stages into the outcome of the process. And then we extract an answer to our question from the full outcome through transformation by a statistic. (See Figure 13.)

Example 8.2 Hunter's Success

Eight hunters each get one shot at a target moving quickly through the woods. All the hunters are “in the zone,” and so their shots are unaffected by the actions of their friends. The hunters have different levels of experience and skill, so for $i \in [0..8)$, the event H_i that hunter i hits the target has Kind

$$\langle \rangle \begin{cases} \text{---} 1 - h_i \text{---} \langle 0 \rangle \\ \text{---} h_i \text{---} \langle 1 \rangle \end{cases}$$

Let the FRP H represent the number of hunters who hit the target. **Find the Kind of H and $\mathbb{E}(H)$, your prediction of the number who hit the target. Also find $\mathbb{E}(\{H > c\})$ for each $c \in \{0, 2, 4\}$, the probability that more than c hunters get a hit.**

Here, we will show two approaches. In the first, we pick specific weights for each hunter and solve the problem for that setting of the weights. In the second, we allow the weights to be variables (symbolic quantities) and solve the problem generally, obtaining answers from this solution for various settings of the weights.

APPROACH #1 SPECIFIC WEIGHTS. We start by assuming one expert hunter, one good hunter, and six novices.

```
h = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.7, 0.9]
hits = [weighted_as(0, 1, weights=[1 - w, w]) for w in h]
```

For $i \in [0..8)$, `h[i]` holds the probability h_i and `hits[i]` equals $\text{kind}(H_i)$. Since all the hunters take independent shots, the number of hits is just the sum of the independent mixture of these:

```
all_hits = independent_mixture(hits)
number_of_hits = Sum(all_hits)
```

The `independent_mixture` function returns the independent mixture of all the Kinds in `hits`, which equal to `hits[0] * hits[1] * hits[2] * hits[3] * hits[4] * hits[5] * hits[6] * hits[7]`.

We apply suitable statistics and take expectations to answer our questions:

```
pgd> number_of_hits
,---- 0.015943 ----- 0
|---- 0.19132 ----- 1
|---- 0.45822 ----- 2
|---- 0.25710 ----- 3
<> -+---- 0.066995 ----- 4
|---- 0.0096001 ----- 5
|---- 0.00078384 ----- 6
|---- 0.000034360 ----- 7
`---- 6.3000E-7 ----- 8

pgd> E(number_of_hits)
2.2

pgd> E(number_of_hits ^ (__ > 4))
0.0104

pgd> E(number_of_hits ^ (__ > 2))
0.3345

pgd> E(number_of_hits ^ (__ > 0))
0.9841
```

The probability of at least one hunter getting a hit is very high; the probability of more than half getting a hit is very low. We predict 2.2 hits, though the probability of getting at least that many hits is substantially less than 1/2. This happens because there is a large probability of getting 1, 2, or 3 hits.

APPROACH #2 SYMBOLIC WEIGHTS. Next, we will mimic this analysis but more generally, letting h contain an unspecified symbolic quantity for each hunter's skill level. We can then define multiple different “profiles” of hunters' skills (e.g., all the same, one expert) and solve our problem for each profile.

```
h = symbols('h_0 ... h_7')
hits = [weighted_as(0, 1, weights=[1 - w, w]) for w in h]

all_50_50 = substitute_with(dict(h_0=0.5, h_1=0.5, h_2=0.5, h_3=0.5,
                                h_4=0.5, h_5=0.5, h_6=0.5, h_7=0.5))
one_expert = substitute_with(dict(h_0=0.1, h_1=0.1, h_2=0.1, h_3=0.1,
                                h_4=0.1, h_5=0.1, h_6=0.1, h_7=0.9))
expert_plus = substitute_with(dict(h_0=0.1, h_1=0.1, h_2=0.1, h_3=0.1,
                                h_4=0.1, h_5=0.1, h_6=0.7, h_7=0.9))
some_elders = substitute_with(dict(h_0=0.1, h_1=0.1, h_2=0.1, h_3=0.1,
                                h_4=0.1, h_5=0.75, h_6=0.75, h_7=0.75))
random_skill = substitute_with({str(w): random() for w in h})

p = symbol('p')
all_equal = substitute_with(dict(h_0=p, h_1=p, h_2=p, h_3=p,
                                h_4=p, h_5=p, h_6=p, h_7=p))
```

The elements of h are “symbols” h_0, h_1, \dots, h_7 which can have numbers substituted for them after we do the calculations. The other variables like `all_equal` hold *functions* that represent hunter profiles; each function substitute the specific numeric values for the symbols in an expression.

Now, we follow the pattern before to get symbolic results:

```
all_hits = independent_mixture(hits)
number_of_hits = Sum(all_hits)
```

We can compute the answers to our questions symbolically, then substitute each profile after the fact.

```
exp_hits = E(number_of_hits)
gt4_hits = E(number_of_hits ^ (__ > 4))
gt2_hits = E(number_of_hits ^ (__ > 2))
gt0_hits = E(number_of_hits ^ (__ > 0))
```

The first of these gives us particular insight; it shows us

```
pgd> exp_hits
h_0 + h_1 + h_2 + h_3 + h_4 + h_5 + h_6 + h_7.
```

The expected number of hits is just the sum of the probabilities that each hunter hits the target. This holds for every profile by the Additivity property and because the expectation of an independent mixture is the tuple of the component expectations, equation (7.16):

$$\begin{aligned}
 \mathbb{E}(H) &= \mathbb{E}(\text{Sum}(H_0 \star H_1 \star \cdots \star H_7)) \\
 &= \text{Sum}(\mathbb{E}(H_0 \star H_1 \star \cdots \star H_7)) \\
 &= \text{Sum}(\langle \mathbb{E}(H_0), \mathbb{E}(H_1), \dots, \mathbb{E}(H_7) \rangle) \\
 &= \mathbb{E}(H_0) + \mathbb{E}(H_1) + \cdots + \mathbb{E}(H_7).
 \end{aligned}$$

The expressions for `gt4_hits` and so forth are much more complicated and harder to parse. But we can recover what we did about by substituting the values from the profiles defined earlier. For example, the profile `expert_plus` matches what we used in Approach #1.

```
pgd> expert_plus(exp_hits)
11/5
pgd> expert_plus(gt4_hits)
0.01041895
pgd> expert_plus(gt2_hits)
0.33451777
pgd> expert_plus(gt0_hits)
0.98405677
```

And as expected, we get the same answers earlier.

For profile `one_expert`, we get somewhat different results

```
pgd> one_expert(exp_hits)
8/5
pgd> one_expert(gt4_hits)
0.00247285
pgd> one_expert(gt2_hits)
0.13729411
pgd> one_expert(gt0_hits)
0.95217031
```

Try computing the results for the other profiles. The one profile that is different is `all_equal`, which replaces eight *different* symbol for the common symbol 'p'. Look at

```
pgd> all_equal(exp_hits)
8 p
pgd> all_equal(gt4_hits)
56 p^5 + -1.4E+2 p^6 + 1.2E+2 p^7 + -35 p^8
pgd> all_equal(gt2_hits)
56 p^3 + -2.1E+2 p^4 + 336 p^5 + -2.8E+2 p^6 + 1.2E+2 p^7 + -21 p^8
pgd> all_equal(gt0_hits)
8 p + -28 p^2 + 56 p^3 + -7E+1 p^4 + 56 p^5 + -28 p^6 + 8 p^7 + -1 p^8
```

and the results are still complex but give us more insight. In fact, if we compare `exp_hits` and `all_equal(exp_hits)` we see that latter gives $8p$. Had there been n hunters instead of eight, we can surmise that the expected number of hits would be np . The probabilities `gt4_hits` et cetera are also simpler polynomials in p in this case, though it takes a bit of simplification to make them digestible (as we will see later on). Note, however, that $\mathbb{E}(\{H > j\})$ has all powers of p from $j + 1$ to 8.

We can ask one more question. The expected number of hits gives our prediction for the number of hits, but does not tell us much about the *consistency* of these results. Consider the profiles `all_50_50` and

```
hot_and_cold = substitute_with(dict(h_0=0.05, h_1=0.05, h_2=0.05, h_3=0.05,
                                     h_4=0.95, h_5=0.95, h_6=0.95, h_7=0.95))
```

Both profiles have 4 as the expected number of hits, but which would be more consistent if the hunters tried on target after target? We define a statistic to answer that question, which gives us the distance between number of hits and 4:

```
@statistic(codim=1)
def spread(num_hits):
    return abs(num_hits - 4)
all_50_50( E(spread(number_of_hits)) )
hot_and_cold( E(spread(number_of_hits)) )
```

The expectation of `spread(number_of_hits)` just computes the MAD measure of uncertainty from Example 7.4; it predicts how far H will be from $\mathbb{E}(H)$. The first profile gives about 1.09 and the second 0.33. Thus, our prediction is that if we repeat the experiment many times, the number of hits will “typically” be about 4 ± 1.09 if all hunters are 50-50 shots but 4 ± 0.33 if half are excellent shots and the others terrible. The consistency of the outcome is stronger in the second case; the values are more “spread” on repetition in the first case.

The pattern in the previous example is that we collect several independent random outcomes and apply a statistic to answer a question. We took the independent mixture of all eight hunters’ attempts and asked a question about those outcomes. (How many hit? Did more than 4 hit?) This emphasizes the role that statistics play in our analysis, which is to express questions that we ask of our data. The next example is a similar instance of this pattern, except it does not entail an *independent* mixture.

Example 8.3 Six of One, Equilateral of the Other

On a regular hexagon with maximum width 2, we choose three distinct vertices randomly so that all subsets of 3 vertices have equal weight.

Define two FRPs:

- A represents the area of the triangle formed by connecting the three vertices.
- Q is the event that the triangle formed by the three vertices is equilateral.

Find the Kind of Q and $\mathbb{E}(Q)$, and find the Kind of A and $\mathbb{E}(A)$.

Imagine that we have labeled the vertices of the hexagon $1, 2, \dots, 6$ in the clockwise direction from one of the vertices (chosen arbitrarily). Picking three distinct vertices means picking a subset of size three from the set $\{1, 2, \dots, 6\}$. There are $\binom{6}{3} = 20$ such subsets.

In the playground, we can use the Kind factory `without_replacement` to find the Kind of a randomly selected subsets where all subsets have equal weight.

```
vertices = without_replacement(3, irange(1, 6))
```

With `size(vertices)`, we can check that there are 20 possible values. These tuples of indices are in the original order (like $\langle 1, 2, 4 \rangle, \langle 2, 5, 6 \rangle$), so we have an equilateral triangle if there is *a gap of exactly 2* between successive numbers in that list. The built-in statistic `Diff` computes the differences between successive components in its input. A condition that tests for gaps of 2 is:

```
is_equilateral = Diff == (2, 2)
```

The Kind of Q is a transform of the Kind of the vertices:

```
equilateral = is_equilateral(vertices)
```

This transformed Kind combines all branches (out of 20) in `vertices` with the same value of the statistic `is_equilateral`, adding their weights. You might find it useful to look at `vertices` and `equilateral` in the playground.

```
pgd> E(equilateral)
1/10
```

yielding $\mathbb{E}(Q) = 1/10$. There are two choices, $\langle 1, 3, 5 \rangle, \langle 2, 4, 6 \rangle$, that satisfy the required condition. To see this, enter

```
vertices ^ Fork(Id, is_equilateral)
```

which will show all the values of `vertices` marked with an extra component that is 1 when the condition is true.

To analyze A , it will help to use Heron's formula: the area of a triangle with side lengths a, b, c is $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$ is the

“semi-perimeter.” This suggests a statistic that maps the three side lengths of the triangle to the triangle’s area:

```
@scalar_statistic
def heron(a, b, c):
    "returns the area of a triangle with given side lengths"
    s = (a + b + c) / 2
    return Sqrt(s * (s - a) * (s - b) * (s - c))
```

We will get the side lengths of the triangle from another statistic.

If the hexagon has width 2, it has side length 1. So, if we pick two vertices that are separated by 1 or 5 in our label, the distance is 1; by 2 or 4, the distance is $\sqrt{3}$; and by 3, the distance is 2.

```
vertex_dists = [0, 1, numeric_sqrt(3), 2, numeric_sqrt(3), 1]
```

```
@statistic
def side_lengths(triangle):
    "computes triangle side lengths from vertex positions on hexagon"
    return [vertex_dists[numeric_abs(triangle[i] - triangle[(i - 1) % 3])]
            for i in range(3)]
```

where the $(i - 1) \% 3$ picks the last index when $i == 0$. Putting this together, we apply both statistics and add a `clean` operation to eliminate round-off error in the calculations, yielding the Kind of A :

```
pgd> area = clean( vertices ^ side_lengths ^ heron )
      ,---- 0.30 ---- 0.43301
<> -+----- 0.6 ----- 0.86603
      `---- 0.1 ----- 1.2990
```

with exact values $\sqrt{3}/4$, $\sqrt{3}/2$, and $3\sqrt{3}/4$. $E(\text{area})$ matches equation (7.29)

$$\mathbb{E}(A) = 0.3\frac{\sqrt{3}}{4} + 0.6\frac{\sqrt{3}}{2} + 0.1\frac{3\sqrt{3}}{4} = 0.45\sqrt{3},$$

as you can confirm in the playground.

Example 8.4 Tournament

Eight players are ranked and play in an elimination tournament. In the first round, player 1 (top ranked) is matched against player 8 (bottom ranked), 2 against 7, 3 against 6, and 4 against 5. The winner of each match advances to the next round with 1 or 8 against 4 or 5 and 2 or 7 against 3 or 6. And so on until only one player remains.

In any given match, if players of rank $r_1 \leq r_2$ are competing, the better seeded player (of rank r_1) is $1.15^{r_2-r_1}$ times more likely to win.

Find the Kind of the rank of the player who wins the tournament and its expectation. Find the probability that a player from the bottom half of the rankings wins the tournament.

We will model this situation by keeping track of the players (by rank) who are still in the tournament. We will arrange their numbers in a tuple such that, at each stage in the tournament, each pair of players who are matched will be adjacent in the tuple. Thus, for the first round, we have

`first_round = constant(1, 8, 4, 5, 2, 7, 3, 6)`

We can see that 1 and 8 are matched, as are 4 and 5, 2 and 7, and 3 and 6. Moreover, who ever wins in the first round will be next to the player they are matched against in the next round. So, 1 or 8 will play 4 or 5. And similarly in the final round.

Next, we need a conditional Kind that takes the slate of players in the current round and produces a Kind for the slate of players in the next round. To do this, we move down the input tuple of players in pairs and for each pair r_1, r_2 produce the Kind

$$\langle \rangle \begin{cases} 1 & \text{---} \langle r_2 \rangle \\ 1.15^{r_2-r_1} & \text{---} \langle r_1 \rangle \end{cases}$$

for the winner of that matchup. (This Kind works regardless of which of r_1, r_2 is smaller.) The independent mixture of these Kinds is then the output.

The conditional Kind for the second round has type $8 \rightarrow 4$, for the third round has type $4 \rightarrow 2$, and for the final round has type $2 \rightarrow 1$. We can implement all of these conditional Kinds in a single function:

```
@conditional_kind
def next_round(players):
    n = len(players) # Always a power of 2 here
    k = Kind.empty
    for i in range(0, n, 2):
        r1, r2 = players[i], players[i + 1]
        odds = as_quantity('1.15') ** (r2 - r1)
        k = k * either(r1, r2, odds)
    return k
```

With this in hand, we can now express the Kind of the next round by *conditioning next_round on the current round*.

```
second_round = next_round // first_round
third_round = next_round // second_round
winner = next_round // third_round
```

Look at these in the playground; they act as we expect. For example, the combination $\langle 1, 4, 2, 3 \rangle$ is most likely to come out of the first round, with $\langle 1, 5, 2, 3 \rangle$ right behind. Similarly, the top ranked players are more likely to win the tournament in the right order. Here's what the Kinds of the third-round matchups and the winner look like:

```
,---- 0.17270 ----- <1, 2>
|---- 0.14158 ----- <1, 3>
|---- 0.075013 ----- <1, 6>
|---- 0.060314 ----- <1, 7>
|---- 0.094949 ----- <4, 2>
|---- 0.077838 ----- <4, 3>
|---- 0.041242 ----- <4, 6>
|---- 0.033160 ----- <4, 7>
```

```

<> -|
|---- 0.076683 ---- <5, 2>
|---- 0.062864 ---- <5, 3>
|---- 0.033308 ---- <5, 6>
|---- 0.026781 ---- <5, 7>
|---- 0.039784 ---- <8, 2>
|---- 0.032614 ---- <8, 3>
|---- 0.017280 ---- <8, 6>
`---- 0.013894 ---- <8, 7>

,---- 0.26520 ----- 1
|---- 0.20843 ----- 2
|---- 0.16017 ----- 3
|---- 0.12058 ----- 4
<> -|
|---- 0.090552 ---- 5
|---- 0.068000 ---- 6
|---- 0.050322 ---- 7
`---- 0.036742 ---- 8

```

We can see for instance that players seeded 1 and 2 have about a 17% probability of meeting in the finals and player 1 has about a 26.5% probability of winning.

To remind yourself how we compute these probabilities, recall that, say, `next_round // third_round` starts by computing the mixture `third_round >> next_round` and then projects out the last component. Each possible slate of players produced in the next round is derived from some slate in the current round through particular outcomes of those match-ups. We compute the probability of that slate by finding all the combination of current round and match-up outcome that produce it (the mixture) and then adding up the weights on all the branches with that slate (the projection). Look at the mixture in the playground and track through the projection to see it.

For example, the winner $\langle 1 \rangle$ can occur from any of the match-ups $\langle 1, 2 \rangle$, $\langle 1, 3 \rangle$, $\langle 1, 6 \rangle$, or $\langle 1, 7 \rangle$ in the current round. Those match-ups happen with probabilities

about 0.173, 0.142, 0.075, 0.060. And in those match-ups the player seeded 1 wins with probabilities $\frac{1.15}{1+1.15}$, $\frac{1.15^2}{1+1.15^2}$, $\frac{1.15^5}{1+1.15^5}$, and $\frac{1.15^6}{1+1.15^6}$, which come from the conditional Kind `next_round`. The result is

$$0.173 \cdot \frac{1.15}{1+1.15} + 0.142 \cdot \frac{1.15^2}{1+1.15^2} + 0.075 \cdot \frac{1.15^5}{1+1.15^5} + 0.060 \cdot \frac{1.15^6}{1+1.15^6} \approx 0.265.$$

The same basic story holds at each stage. See page 189.

Now, we can answer our questions

```
pgd> E(winner)
3.151837905582001
pgd> E(winner ~ (__ > 4))
0.2456155413215654
pgd> E(winner ~ (__ == 1))
0.2652034184779578
```

which give values of about 3.15 and just under 0.25, respectively. The top-seeded player has about a 26.5% probability of winning the tournament, but the other players – especially the 2 and 3 and 4 seeds – have a non-trivial chance of winning as well. So our prediction for the winning seed is just over 3.

The next example illustrates the important concept of *state* that describes the configuration of a random system at a given moment. (See Section 6.2.) We need enough information in the state to understand the future evolution of the system at any point but not so much information that it obscures the essentials. Crafting a good description of state for a process is part art and part science, and we will get lots of practice. As you read the next example, think about how you would describe the state of the mouse’s process.

Example 8.5 Mouse Escape

A mouse is caught in a room with a cat and wants desperately to escape. The room has been newly refurbished, so the usual escape routes have been plugged. The only hope is for the mouse to climb three steps to safety.

The mouse starts on the floor (step 0) and successively attempts to climb onto the next step. If the mouse fails it tumbles down to the *previous* step

(except on step 0); if it succeeds, it moves to the next step and tries again. So for example, if it fails to climb to step 2 from step 1, it falls to step 0; if it succeeds, it moves to step 2.

If the mouse reaches step 3 by the 16th attempt, it escapes; otherwise the cat eats it. On each attempt the Kind of whether it succeeds or fails has twice the weight on failure.

Let M be the event that the mouse escapes. Find $\mathbb{E}(M)$, the probability that the mouse escapes.

The process by which the mouse tries to escape from its first attempt to its eventual escape or capture can be described by a state that evolves from attempt to attempt. What do we need to keep track of to be able to model the evolution of the process and answer the questions we care about?

First, at any point, we need to know which step the mouse is on, 0, 1, 2, or 3. If we know where the mouse is, we can determine the possibilities for its subsequent attempts. Second, we also need to know how many attempts the mouse has made because if it takes too many the cat will catch it.

We have two choices. We can set the state as the step the mouse is on and account for the possibility of capture by analyzing whether it escapes by the 16th attempt, or we can include the number of attempts in the state as well. We will illustrate both approaches here.

If the state is just the step the mouse is on, then the initial state is 0, and the mouse moves up or down (or stays at 0) with twice the weight on down.

```
initial_state = constant(0)
move = conditional_kind({
    0: either(0, 1, 2),
    1: either(0, 2, 2),
    2: either(1, 3, 2),
    3: constant(3)
})
```

The first is the Kind of the initial state, and the second is the conditional Kind of the next state *given* the initial state. For instance, from step 0, the mouse

either stays at 0 or moves to 1 with twice the chance of staying as moving up. If the mouse is at step 3, we model it as staying there; it has escaped. If it has reached state 3 by the 16th attempt, we know that it escaped on or before that attempt.

We can compute this by starting in the initial state and iterating for 16 attempts:

```
state = initial_state
for _ in range(16):
    state = move // state
escaped = E(state ^ (__ == 3))
```

The operation `move // state` is *conditioning*, specifically, we are finding the Kind of the next state by conditioning on the current state. The conditional Kind `move` gives the next state's Kind for each value of the current state, but it does not give any information about how likely any value of the state is. We can view the conditioning operation as computing the Kind of the next state as a weighted average of the Kinds `move(s)` over all values of `state` using its canonical weights. (The loop above is implemented by the builtin playground function `evolve` with `evolve(initial_state, move, 16)`.)

Our question is answered by the expectation `escaped`, which is approximately 0.368. This is the probability that the mouse escapes before the cat catches it. In this approach we account for the mouse's time horizon by evolving the system over 16 moves.

For the second approach, we use an expanded definition of state that keeps track of the number of moves and freezes the state when either the mouse escapes or the cat eats it. This approach is slightly more complicated, but it illustrates how we can include contingent dynamics in our systems.

Now, the state is a tuple $\langle n, s \rangle$, where n is a number of attempts and s is either a step (0, 1, 2, 3) or -1 to indicate that the mouse has been captured. The initial state is $\langle 0, 0 \rangle$, and we have

```
initial_state_alt = constant(0, 0)
```

Our state transitions now depend on how many moves the mouse has made. If

the mouse has escaped or been eaten, we simply keep the state unchanged. If it is the 16th attempt, it will be captured if it doesn't make it to step 3. Otherwise, the mouse moves as before and the number of attempts is incremented.

```
@conditional_kind
def move_alt(attempts_and_step):
    n_attempts, step = attempts_and_step

    # If we are at the end, stay there
    if step == 3 or step == -1:
        return constant(attempts_and_step)

    n = n_attempts + 1

    # If the cat is here, last chance
    if n_attempts == 16:
        if step < 2:
            return constant(n_attempts, -1)
        else:
            return either((n, -1), (n, 3), 2)

    # From step 0, we either stay or move up
    if step == 0:
        return either((n, 0), (n, 1), 2)

    # Otherwise, we move up or down
    return either((n, step - 1), (n, step + 1), 2)
```

To answer our questions, we evolve the system and transform the resulting Kind as we did earlier, except the mouse's outcome is in the second component.

```
mouse_outcome = evolve(initial_state_alt, move_alt, 16)
escaped_alt = E(mouse_outcome ^ (Proj[2] == 3))
```

We evolve the system 16 steps because we need not do more, bbut evolving it for

longer would not change the result because the state becomes fixed.

8.3 Strategies and Representations

Despite their power, neither computation nor mathematics are fully “automatic.” We often need to apply some creativity to get useful results, either to make a computation feasible/efficient or to make a mathematical analysis tractable. In this subsection, we look at examples where we bring a little flare to select our strategies or data representations.

These examples demonstrate some common patterns in how we approach more complicated systems. The next example illustrates the pattern of optimizing a decision over a menu of strategies for making that decision.

Example 8.6 Assistant Assistance

You are trying to hire an assistant to help you with your work, so you place an ad in the local paper. The next day, exactly n applicants call you to schedule an appointment for an interview, and you schedule them at random in n time slots. Assume that all ordering of the appointments are equally likely, i.e., have the same weight.

You have a very particular set of criteria in mind for the position. At each interview, you evaluate the applicant’s qualifications with respect to these criteria, so after each interview, you can unambiguously rank the applicants you’ve seen up to that point. However, the job market is running hot, and if you do not offer the job to an applicant immediately after the interview, the applicant will take another job and is lost to you.

Our task is to pick a good strategy for deciding whether to offer an applicant the job and to assess how well that strategy performs. We will consider the family of strategies in which you reject the first k applicants and then offer the job to the first applicant after these that ranks better than all of those first k . There are n different strategies here, which we call “After 0”, “After 1”, “After 2”, ..., “After $n - 1$ ”. In the “After 0” strategy you would always choose the first applicant. With “After k ” for $k > 0$, it is possible to make no offers. We will say that our strategy has *succeeded* if we make an offer to the overall best-ranked applicant.

Find the probability that the “After k ” strategy succeeds for each $k \in [0..n)$. Which of these strategies is most likely to succeed?

We start by reasoning to reduce the problem to a simpler form in two steps. First, we only need to keep track of the positions of two applicants: the “best” applicant who has the best rank overall and the “pre-best” applicant who has the best rank among all the applicants who appear *before* the best applicant. The “After k ” strategy succeeds in finding the best applicant if the best applicant appears in position b with $b > k$ and the pre-best applicant appears in position s with $s \leq k$. If $b = 1$, there is no pre-best applicant, and we set $s = 0$. (If $b \leq k$, the best applicant would be skipped. If $k < s < b$, we would choose the pre-best applicant who has higher rank than all those at positions $[1..k]$.)

Second, because all orderings of the applicants are equally likely,

- (i) the best applicant is equally likely to be in any position, and
- (ii) *given* that the best applicant appears in position $b > 1$, the position of the pre-best applicant is equally likely to be in any position in $[1..b-1]$.

Claim (i) is true because the Kind for the permutation of the n ranks has size $n!$ with all equal weights. So for any of the n positions of the best applicant, there are $(n-1)!$ branches permuting the other weights, so the position of the best applicant has weight $1/n$ on each possible value. Claim (ii) is true by a similar argument; once we fix the position of the best applicant, the remaining branches iterate over all orderings of ranks that fit before the best applicant, all of which have equal weight. Using the code from this example, you can see this in the playground by entering `check_best_position()` to see the Kind of the best position when $n = 7$ and `check_pre_best_position(b)` to see the Kind of the pre-best position (when $n = 7$) for each best position.

The direct but inefficient way to solve this problem would be to construct the Kind of all orderings of the n rankings, transform this with a statistic that extracts the best and pre-best positions, and then transform with a condition that $b > k$ and $s \leq k$. In the playground, for specific values of n and k , this would look like

```
permutations_of(irange(n)) ^ best_pre_best_of ^ is_success(k)
```

using the statistics `best_pre_best_of` and statistic factory `is_success` defined in the example code, e.g.,

```
def is_success(k):
    "Statistic factory that checks if After-k succeeds given <b,s>."
    @condition
    def succeeded(b, s):
        return b > k and s <= k

    return succeeded
```

For small n , this works fine, but the size of the initial Kind grows quickly.

Instead, we will use our earlier reasoning to directly derive the Kind of the best and pre-best positions. To allow this to work for any n , we wrap the whole analysis in a function that takes n as a parameter:

```
def assistant(n):
    "Returns success probabilities of After-k strategies with `n` applicants."
    assert n >= 1, "at least one applicant is required"

    best = uniform(irange(n))

    @conditional_kind(codim=1)
    def pre_best_position(m):
        if m <= 1:
            return constant(0)
        return uniform(irange(1, m - 1))

    best_pre_best = best >> pre_best_position

    success_probs = [0] * n
    for k in range(n):
        success_probs[k] = as_scalar( E(best_pre_best ^ is_success(k)) )
```

```
return as_vec_tuple(success_probs)
```

(The application of `as_scalar` to the expectation unwraps the number from the tuple.) Here is how we use it:

```
pgd> assistant(7)
<0.14285714285714286, 0.35, 0.41428571428571423, 0.4071428571428571,
  0.35238095238095238, 0.2619047619047619, 0.14285714285714286>

# The direct approach gives the same answers, compare k=2 and k=3
pgd> E(permutations_of(irange(7)) ~ best_pre_best_of ~ Fork(is_success(2), is_success(3)))
<0.4142857142857143, 0.4071428571428571>

pgd> best_k(assistant(7)) # Find k that maximizes the probability
<2, 0.4142857142857143>
```

The function `best_k` is defined in the example code; it returns the best k and its success probability for the specified n .

Trying this for various n , we see the optimal After- k strategy, k^* , has $k^* \approx ne^{-1}$ with a probability approaching $e^{-1} \approx 0.36788$ for large n .

n	k^*	$\lfloor \frac{n}{e} \rfloor$	$\lceil \frac{n}{e} \rceil$	Probability
10	3	3	4	0.39869
20	7	7	8	0.38421
30	11	11	12	0.37865
40	15	14	15	0.37574
50	18	18	19	0.37428
60	22	22	23	0.37321
70	26	25	26	0.37239
80	29	29	30	0.37186
90	33	33	34	0.37142
100	37	36	37	0.37104
250	92	91	92	0.36915
500	184	183	184	0.36851
1000	368	367	368	0.36820

We will see later where this comes from.

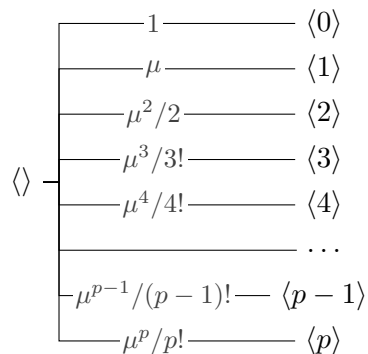
The next example uses a carefully chosen, but somewhat elaborate, representation of the system's state to make it possible – in fact, straightforward – to compute the answer we seek. The choice of representation still needs to keep track of the essential information, but by discarding everything else, computations that would be slow become manageable.

Example 8.7 Elevator Stops

An elevator opens on the ground floor and a random number of passengers enter it. Each passenger selects one floor (above the ground floor), and the elevator proceeds upward, stopping at each selected floor.

- There are n floors above the ground floor.
- The FRP N represents the number of passengers entering the elevator. Its Kind of N is given below. It depends on numeric parameters p and μ and is given below.
- Each passenger's chosen floor is represented by an FRP with Kind `uniform(1, 2, ..., n)`, with all floors equally likely.
- All passenger choices are independent of each other.

If no passengers enter, then the elevator makes zero stops. The FRP S represents the number of stops the elevator makes. For any positive integer p and $\mu > 0$, the Kind of N is



The parameter p is the maximum number of passengers allowed in the elevator, and μ determines the weights. In the example code, the function `passengers_kind(p, mu)` computes this Kind in canonical form. **Take** $n = 10$, $p = 21$, and $\mu = 5$. **Find** $\mathbb{E}(S)$.

With 20 or more passengers possible, the Kinds we need will be slow to compute because there are 10 possible floors for each passenger. Our first insight is that keeping track of the floor each passenger visits is not really necessary. We only need to keep track of *which floors are visited*, whether they are visited by one or many passengers is irrelevant to our needs.

Thus, the state we use for describing this system is **the set of visited floors**. With $n = 10$, we encode this set as a 10-tuples containing only 0's and 1s; a one in slot i means that floor i is visited by at least one passenger and a 0 means that floor is not visited by any passengers. The Kind for the set of visited floors with this representation has size ≤ 1024 , which is manageable. (It does grow quickly with n , but we are keeping n fixed here.)

If we are going to use 10-tuples of bits to represent the set of visited floors, we need two operations on these sets: (1) converting a list of requested floors to a set in our representation, and (2) combining two sets into one, taking a 20-tuple that encodes two sets (10 components each) and returning a single 10-tuple representing their union. These operations are given, respectively, by the statistics `visited_floors(10)` and `union_visited(10)`. These are both nice examples of how we can use statistics to convert data from one representation to another. We implement these as *statistic factories* that take the index of the top floor and return the statistic we seek for that building. Here, we will only be using these functions with an argument of 10.

```
def visited_floors(top_floor: int) -> Statistic:
    "Returns a statistic converting a list of floor choices to a set."
    @statistic(name=f'visited_floors<{top_floor}>')
    def visited_set(value):
        "returns the set of unique components in a fixed range, as a bit string"
        bits = [0] * top_floor
        for x in value:          # values are floors in 1, 2, ..., top_floor
            bits[x - 1] = 1      # this floor's button has been pushed
        return as_vec_tuple(bits)
    return visited_set
```

```

def union_visited(top_floor: int) -> Statistic:
    "Returns a statistic that unions two `top-floor` sets as bit-strings."
    @statistic(name=f'union<{top_floor}>', monoidal=as_vec_tuple([0] * top_floor))
    def union(value):
        "unions two `top_floor`-length bit strings into one with a bitwise-or"
        return as_vec_tuple(value[i] | value[i + 10] for i in range(top_floor))
    return union

max_floor = 10
as_set = visited_floors(max_floor) # Statistic: convert floors to visited sets
union = union_visited(max_floor)   # Statistic: Union of two sets as 10-bit strings

```

Now we are ready for our analysis. First, we build a conditional Kind `floors` that maps the number of passengers to the Kind of the visited floor set for that many passengers. We do this iteratively, by mixing in the choices of one new passenger to our previously computed Kinds.

```

passenger_floor = uniform(1, 2, ..., max_floor)
choice = as_set(passenger_floor) # Kind for each floor choice as set
floors = {0: constant([0] * max_floor), 1: choice, 2: union(choice * choice)}
for i in irange(2, 20):
    floors[i + 1] = union(floors[i] * floors[1])
visited = conditional_kind(floors) # kind of visited floor set for each # pass.

```

Each element in the loop adds a new entry to the conditional Kind that accounts for an extra passenger's choices in the set of visited floors. You should look at a few of these Kinds in the playground.

If we apply the `Sum` statistic to this conditional Kind, it transforms it to a new conditional Kind that maps the number of passengers to the Kind of the *number of visited floors*. (Make sure you see why.) All that remains is to mix in the number of passengers. We use the conditioning operator `//` because we want to average over the various numbers of passengers without retaining it in the result.

```
number_passengers = passengers_kind(21, 5)
number_visited_floors = Sum(visited) // number_passengers
```

The Kind `number_visited_floors` looks like

```
pgd> number_visited_floors
,---- 0.0067379 ----- 0
|---- 0.043710 ----- 1
|---- 0.12760 ----- 2
|---- 0.22074 ----- 3
|---- 0.25060 ----- 4
<> -+---- 0.19508 ----- 5
|---- 0.10546 ----- 6
|---- 0.039095 ----- 7
|---- 0.0095105 ----- 8
|---- 0.0013710 ----- 9
`---- 0.000088937 ----- 10
pgd> E(number_visited_floors)
3.934693309902328
```

So, $\mathbb{E}(S) \approx 3.935$. In the example code, this analysis to get the Kind `number_visited_floors` is run by calling `elevator_stops(max_floors=10, p=21, mu=5)`.

There were a lot of moving parts in the last example, but the key feature was changing the representation of the system from a list of floors requested/visited by passengers in the elevator to a fixed-size set indicating which floors have been visited. With this representation, we could account for each added passenger with an independent mixture followed by a statistic that combines the two sets. This is a complicated example of a common pattern.

Example 8.8 Rig at Risk

A mid-ocean oil rig accumulates damage from severe waves over time. Assume that in a given year, the number of severe waves is given by the value of an FRP N and that the damage caused by severe wave i is given by the output of FRP D_i , where all the D_i 's have the same Kind. Assume that the D_i 's are independent of

N . Let T be the FRP representing the total damage that accumulates in a year.

Find $\mathbb{E}(T \mid N = n)$, the expectation of the total damage during the year given that there were n severe waves, and $\mathbb{E}(T)$.

There are two things to specify in this problem: the Kinds of N and D . But we can do the analysis and get some insight by doing the analysis in terms of these unspecified Kinds.

The first point to recognize is that the structure is simple when we know the value of N . Specifically, *given* that there are n waves, the Kind of the total damage can be expressed in two steps:

1. take an independent mixture of n copies of D_1 's Kind, and
2. apply the **Sum** statistic.

That is,

$$\text{kind}(T \mid N = n) = \text{Sum}(\text{kind}(D_1) \star \star n).$$

On the left, we have a conditional Kind (given the value n of N) and the right an exact expression for it. By the Additivity property (7.12) of expectation,

$$\mathbb{E}(\text{Sum}(\text{kind}(D_1) \star \star n)) = \mathbb{E}(D_1) + \mathbb{E}(D_2) + \cdots + \mathbb{E}(D_n),$$

and because all D_i 's have the same Kind and expectation,

$$\mathbb{E}(T \mid N = n) = n \mathbb{E}(D_1).$$

So viewing $\text{kind}(T \mid N = n)$ as a conditional Kind, we can *condition on* N :

$$\text{kind}(T \mid N = n) // \text{kind}(N)$$

See page 189.

and take expectations. As we've seen earlier, this operation averages the Kinds $\text{kind}(T \mid N = n)$ over n weighting by $\text{kind}(N)$. The expectation is therefore

$$\mathbb{E}(T) = \sum_n p_n \mathbb{E}(T \mid N = n), \quad (*)$$

where p_n is the weight on n in $\text{kind}(N)$.

The function `rig_at_risk` in the example code implements this in `frplib`.

```
def rig_at_risk(kind_N, kind_D):  
    """Computes expectation of total wave damage by mixing kinds then computing E.  
    Parameters `kind_N` and `kind_D` give arbitrary kinds for the number of waves  
    and the damage per wave.  
  
    """  
    @conditional_kind(codim=1)  
    def damage_given_n(n):  
        return fast_mixture_pow(Sum, kind_D, n)  
  
    return E(damage_given_n // kind_N)
```

For instance:

```
pgd> rig_at_risk(uniform(1, 2, ..., 10), uniform(1, 2, 3))  
11
```

You can play with different input Kinds using the example code.

8.4 Using Observations

In general, we build our model and compute our predictions before the random process we are studying begins. But we allow for making decisions or interventions as the process unfolds, so we need the ability to *update our predictions* in light of new information about uncertain quantities. This is the role of conditional constraints.

The results of such updates can seem counter-intuitive because they balance multiple possibilities. The easiest way to handle this is to remember that constraining with conditionals simply eliminates the branches of the Kind that are inconsistent with our observations. When we renormalize into canonical form, the numbers change, but the relative sizes of the remaining branches' weights *do not change*. Put more glibly: probability does not move when we update our predictions.

The next example is gleefully counter-intuitive. We will do the calculations in the playground and then try to understand them.

Example 8.9 A Kid Named “Florida”

We consider two questions

- (a) You know that a neighbor’s family has two children, but you cannot remember whether the children are boys or girls. One day, you see that one of the children is a girl.
- (b) Suppose that during the previous experiment we learn that one of the neighbor’s two children is a girl whose name is very rare, for instance “Florida.” (Mlodinow, 2008)

What is the probability that both children are girls in each situation?

It seems strange that the rarity of the names could have much of an effect, but we’ll see that it does. Let’s solve both parts and then regroup to understand the results.

First, we consider the various outcomes that might occur, ignoring the childrens’ names. The question we want to answer and the information that we observe are represented as statistics:

```
at_least_one_girl = Or(Proj[1] == 1, Proj[2] == 1)
both_girls = And(Proj[1] == 1, Proj[2] == 1)
```

We define the Kind for an event that an individual child is a girl:

```
girl = either(0, 1)
```

Then we look at the Kinds for the various outcomes of interest: whether each of two children are girls and whether each of two children are girls given that at least one is.

```
pgd> girl * girl
,---- 1/4 ---- <0, 0>
|---- 1/4 ---- <0, 1>
<> -|
|---- 1/4 ---- <1, 0>
`---- 1/4 ---- <1, 1>
```

```
pgd> outcome_no_names = girl * girl | at_least_one_girl
pgd> outcome_no_names
,---- 1/3 ---- <0, 1>
<> -+---- 1/3 ---- <1, 0>
`---- 1/3 ---- <1, 1>
```

The conditional constraint eliminates the $\langle 0, 0 \rangle$ branch. Note that the relative size of the weights in the remaining branches *does not change*. Dropping the branch gives three branches with weights $1/4$, and when we re-normalize weights $1/3$. The information that there is at least one girl has simply ruled out the possibility that neither child is a girl. And hence:

```
pgd> both_girls(outcome_no_names)
,---- 2/3 ---- 0
<> -|
`---- 1/3 ---- 1
```

and $E(\text{both_girls}(\text{outcome_no_names})) = 1/3$. is the desired probability in situation (a).

The information we have in the second situation is somewhat different; we know that at least one of the children is a girl with a rare name. So we need, for each child, events that the child is a girl and that the child has a rare name. We encode the latter events with a Kind with a symbolic weight and use arbitrary values 10 and 11 (rather than 0 and 1) to make it easier to identify the components in the values. A rare name means that p is a *small* number.

```
p = symbol('p')
rare = weighted_as(10, 11, weights=[1 - p, p])
neighbors = girl * girl * rare * rare
```

By using a independent mixture for **neighbors**, we are assuming that the rarity of each child's name is independent of whether the child is a girl, as well as assuming that the two children's outcomes are independent. As above, we use a statistic to represent our conditional constraint:

```

a_rare_girl = Or(
    And(Proj[1] == 1, Proj[3] == 11),
    And(Proj[2] == 1, Proj[4] == 11)
)

```

The Kind of interest then becomes

```

pgd> outcome = neighbors | a_rare_girl
,---- (-1 + p)/(-4 + p) ---- <0, 1, 10, 11>
|---- -1 p/(-4 + p) ----- <0, 1, 11, 11>
|---- (-1 + p)/(-4 + p) ---- <1, 0, 11, 10>
<> -+---- -1 p/(-4 + p) ----- <1, 0, 11, 11>
|---- (-1 + p)/(-4 + p) ---- <1, 1, 10, 11>
|---- (-1 + p)/(-4 + p) ---- <1, 1, 11, 10>
`---- -1 p/(-4 + p) ----- <1, 1, 11, 11>
pgd> both_girls(outcome)
,---- 1/(2 + -0.5 p) ----- 0
<> -|
`---- (-2 + p)/(-4 + p) ---- 1

```

Taking expectations $E(\text{both_girls}(\text{outcome}))$, we get $\frac{2-p}{4-p}$. Because the name is rare – that is, p is small – $\frac{2-p}{4-p} \approx \frac{1}{2}$ and substantially bigger than $1/3$.

This is a surprising difference! We can get insight into this result by studying the Kind outcome. Since p is small, we can take it, for this purpose, to be so small that we can ignore it:

```

pgd> clean(substitution(outcome, p = 0))
,---- 1/4 ---- <0, 1, 10, 11>
|---- 1/4 ---- <1, 0, 11, 10>
<> -|
|---- 1/4 ---- <1, 1, 10, 11>
`---- 1/4 ---- <1, 1, 11, 10>

```

This Kind reveals what has happened. We are *very unlikely* to see *two* rare names, but the one rare name can be for *either* of the two girls. When only one

of the children is a girl, the condition can be satisfied in *just one way*. Hence, instead of one out of three possibilities, we have two out of four.

Remember that when we constrain with a conditional, we eliminate branches that are inconsistent with the condition, but the *relative sizes of the weights on the remaining branches does not change*.

Example 8.10 Buckets and Balls

We have two buckets. In the left bucket, there are three red, six blue, and one green ball. In the right bucket, there are four red, four blue, and two green balls. The balls in both buckets are well mixed.

I choose a bucket at random, with equal weights on both, and then from that bucket choose a ball at random, again with equal weights on every ball. You do not see what bucket I chose the ball from, but I show you that I picked a green ball. What is the probability that I chose from the right bucket given this information?

We have a system that is most easily described in two stages. First, I pick a bucket. Second, I pick a ball from the chosen bucket. This is a mixture.

Let's assign 0 to the left bucket and 1 to the right; and let 0 and 1 stand also for not-green and green.

```
bucket = either(0, 1)
green_given_bucket = conditional_kind({
  0: either(0, 1, 9),
  1: either(0, 1, 4)
})
```

For the latter Kinds, the left bucket has 9 not-green balls and 1 green ball, a ratio of 9, and the right bucket has 8 not-green balls and 2 green balls, a ratio of 4. This explains the third argument to `either` in both cases. The FRP that represents the chosen bucket *given* that we have observed a green ball has Kind

```
which_bucket_g = Proj[1]( bucket >> green_given_bucket | (Proj[2] == 1) )
```

We get this in three stages: 1. use a mixture to build the combined Kind of

bucket and chosen ball, 2. apply the constraint that we observed a green ball with a conditional, and 3. extract the first component (the bucket). We can write this more concisely as

```
which_bucket_g = bayes(observed_y=1, x=bucket, y_given_x=green_given_bucket)
```

We get $E(\text{which_bucket_g}) = 2/3$, the probability that we chose the right bucket given that we observe a green ball. Interestingly, we can also find

```
which_bucket_n = bayes(observed_y=0, x=bucket, y_given_x=green_given_bucket)
```

and $E(\text{which_bucket_n}) = 8/17$, the probability that we chose the right bucket given that we observe a not-green ball.

The previous example is a demonstration of **Bayes's Rule**, a common and important pattern we also saw earlier. We have two FRPs X and Y of arbitrary dimension. We know the Kind of X , and we know the *conditional* Kind of Y given X . We then *observe* Y and want to *infer* X . In the previous example, X represents to the bucket we choose the ball from, and Y is the event that we pick a green ball. We build our model for this system with a mixture because it is easier to specify the Kind of ball we pick once we know the bucket.

We know $\text{kind}(X)$, and for any possible value u of X ; we know $\text{kind}(Y \mid X = u)$; and we have observed a value v of Y . Note that the mapping from u to $\text{kind}(Y \mid X = u)$ is a *conditional kind* that we can denote by $\text{kind}(Y \mid X)$.

Bayes's Rule is equivalent to three steps:

1. Build with a mixture the Kind of the combined outcome $\langle X, Y \rangle$:

$$\text{kind}(\langle X, Y \rangle) = \text{kind}(X) \triangleright \text{kind}(Y \mid X).$$

2. Constrain this with a conditional using the observation that $Y = v$:

$$\text{kind}(\langle X, Y \rangle) \mid Y = v$$

3. Transform with a projection statistic to extract the X components $1, \dots, \text{dim}(X)$:

$$\text{proj}_{1..\text{dim}(X)} (\text{kind}(\langle X, Y \rangle) \mid Y = v).$$

The first step gives the combined Kind for X and Y ; the second applies the constraint from our observation; and the third isolates the Kind of X , since that is what we want to know (and we have already observed Y 's value).

In the playground, we can implement these same steps easily. Letting `x` and `y_given_x` stand for $\text{kind}(X)$ and $\text{kind}(Y \mid X)$, the built-in `bayes` method looks like

```
def bayes(observed_y, x, y_given_x):
    i = dim(x) + 1
    return (x >> y_given_x | (Proj[i:] == observed_y)) ^ Proj[1:i]
```

The `bayes` function will work just as well with FRPs as with Kinds; in that case, `x` would be the FRP `X`, and `y_given_x` would be the conditional FRP `Y_given_X`.

Let's use this again, generalizing an earlier example.

Example 8.11 Disease Testing Redux

A disease is prevalent in the population where in any large sample of people, a proportion around d will have the disease. A test has been developed to detect the disease. If a tested patient does *not* have the disease, the test will indicate they are negative with probability n . If a tested patient *does* have the disease, the test will indicate they are positive with probability p .

If a doctor sees that a patient has tested positive, what is the probability that the patient has the disease?

Let D be the event that the patient has the disease and T be the event that they tested positive. First, let's use the information provided to create $\text{kind}(D)$ and $\text{kind}(T \mid D)$, the conditional Kind of T given the observed value of D . These derive directly from the described assumptions.

```
d = symbol('d')
n = symbol('n')
p = symbol('p')

has_disease = weighted_as(0, 1, weights=[1 - d, d])
test_by_status = conditional_kind({
    0: weighted_as(0, 1, weights=[n, 1 - n]),
    1: weighted_as(0, 1, weights=[1 - p, p])
})
```

})

To find the probability that D occurs, we apply Bayes's Rule and take expectations:

```
pgd> disease_given_positive = bayes(1, has_disease, test_by_status)
pgd> disease_given_negative = bayes(0, has_disease, test_by_status)
pgd> E(disease_given_positive)
```

The result is

$$\frac{pd}{pd + (1 - n)(1 - d)}. \quad (*)$$

We can understand this by looking at the combined Kind of $\langle D, T \rangle$:

```
pgd> has_disease >> test_by_status
,---- n (1 - d) ----- <0, 0>
|---- (1 - d) (1 - n) ----- <0, 1>
<> -|
|---- (1 - p) d ----- <1, 0>
`---- p d ----- <1, 1>
```

The conditional constraint that the test is positive eliminates the first and third branch of this Kind, giving (non-canonical) Kind

```
,---- (1 - d) (1 - n) ----- <0, 1>
<> -|
`---- p d ----- <1, 1>
```

The remaining branches represent the possibilities that the patient tests positive and has the disease or that the patient tests positive and does not have the disease. The probability in (*) is just the normalized weight on the second branch.

We can examine these probabilities for various specific values to get a feel for how Bayes's Rule balances the prevalence – or *base rate* – of the disease in the population and the information about the sensitivity and specificity of the diagnostic test.

```

substitution(E(disease_given_positive),
             d='1/1000', n='99/100', p='95/100')
# is 0.0868
substitution(E(disease_given_positive),
             d='1/10_000', n='99/100', p='95/100')
# is 0.0094
substitution(E(disease_given_positive),
             d='1/10_000', n='999/1000', p='999/1000')
# is 0.0908
substitution(E(disease_given_positive),
             d='1/10_000', n='95/100', p='9/10')
# is 0.0018
substitution(E(disease_given_positive),
             d='1/100', n='95/100', p='9/100')
# is 0.1538
substitution(E(disease_given_positive),
             d='1/100', n='999/1000', p='999/1000')
# is 0.9098

```

The first thing to notice is that, except in the last case, the probability of the patient having the disease is quite low, *even with a positive test* and even when the test is highly accurate. The reason is that in these cases the base rate of the disease is low, and as we saw above, the probability accounts for the two possibilities: patient has the disease and tests positive versus does not have the disease and tests positive. Only when the disease is somewhat common and the tests accurate do we get a high probability.

8.5 Touching Infinity

FRPs are particularly suited as models of *finite* random processes: a finite number of possibilities, finite dimension, and a finite horizon of time and space. But in some cases, we can push beyond the finite and get exact results for more general processes.

Example 8.12 Waiting for Heads

We flip a coin repeatedly up to and including the first flip on which a heads comes up. How many flips will it take?

Let H be the FRP representing the number of flips required up to and including the first heads. We can ask several questions about H :

- What is a typical number of flips, $\mathbb{E}(H)$?
- How likely are we to need more than n flips, $\mathbb{E}(\{H > n\})$?
- How likely are we to need exactly n flips, $\mathbb{E}(\{H = n\})$?
- How likely are all the different possibilities, $\text{kind}(H)$?

As we did earlier, we use $\{H > n\}$ and $\{H = n\}$ to denote the events (0-1-valued FRPs) that H is bigger than n and equal to n .

We model 0 as “tails” and 1 as “heads.” We will assume that the coin has the same chance q of coming up tails on any flip and that the outcome of any flip has no influence on the outcome of any other, i.e., they flips are *independent*.

Think of this as a random system like we discussed in Section 6. At any point, we can be in one of two states: we’ve seen a heads or we have not. Call these states HEADS and NO HEADS. The system starts in state NO HEADS. On any flip, if we get a heads, we have made one flip and transition to state HEADS, and if we get a tails, we have made one flip and remain in the same NO HEADS state as before the flip. Here’s the key insight: if we get a tails, the Kind of our *remaining* number of flips until we see a heads is the *same* as $\text{kind}(H)$.

We can make this idea precise with a conditional Kind. Suppose `remaining_flips` is the *Kind* of the number *additional* of flips until a heads *after* the current flip. Then, the Kind of the total number of flips *given* the current flip is

```
conditional_kind({
  0: remaining_flips ^ (__ + 1),
  1: constant(1)
})
```

It takes one flip if we get a heads on the current flip, or one flip – for the current flip – plus the remaining flips required if we get a tails on the current flip.

We write a function `wait_for_heads` that gives the Kind of total number of flips when we have specified the Kind of the remaining number of flips after the current flip.

```
def wait_for_heads(remaining_flips, q=symbol('q')):
    """Returns the Kind of the total number of flips to get a head.

    `remaining_flips` is the kind of the additional number of flips
        required after seeing a tails

    `q` is the probability of getting a tails. [Default: symbol('q')]

    """
    # Make sure q is a symbol or high-precision number
    q = as_quantity(q)

    # The kind of the current flip
    flip = weighted_as(0, 1, weights=[q, 1 - q])

    # the kind of the total number of flips given the current flip
    flips_given_current = conditional_kind({
        0: remaining_flips ^ (__ + 1),
        1: constant(1)
    })

    total_flips = flips_given_current // flip
    return total_flips
```

We do not know `remaining_flips`, which is what we want to find, but we have a trick up our sleeve. The solution (`kind(H)`) is the Kind `K` that equals `wait_for_heads(K)`. It is a “fixed point” of the function `wait_for_heads`.

We can solve for that fixed point as in Section 6, and will below, but first we

will compute an approximation. We start with a guess K_0 for $\text{kind}(H)$ and set $K_1 = \text{wait_for_heads}(K_0)$. Then we use K_1 as our next guess, and continue to iterate until the Kinds we get stop changing to numerical precision. The result will be $\text{kind}(H)$ to good approximation, even though that Kind will have an infinite number of possible values.

In the playground, this looks like

```
guess = constant(1)
guess = wait_for_heads(guess)
guess = wait_for_heads(guess)
guess = wait_for_heads(guess)
guess = wait_for_heads(guess)
guess = wait_for_heads(guess)
```

Each time, we enhance the tree by accounting for an extra possible flip to get what we want, and but only the two branches with the highest values change at each iteration. Compute these one at a time, and look at `guess` at each stage to see how the trees grow.

The sequence of Kinds produced above is what we get using the `iterate` utility in `frplib`. For instance, the third and sixth values of `guess` above equal, respectively, `iterate(wait_for_heads, 2, constant(1))` and `iterate(wait_for_heads, 5, constant(1))`. We can iterate as long as we like: try `iterate(wait_for_heads, 20, constant(1))`. At each iteration, only the largest two branches change.

Using this, we can compute exact probabilities of waiting *more than n flips* and of waiting *exactly n flips*. We iterate a little more than n times, and the part of the tree we need is exact.

```
def wait_more_than(n, q=symbol('q')):
    "Returns probability of waiting more than n flips for heads; q is the weight on tails."
    wait = iterate(wait_for_heads, n + 1, constant(1), q=q)
    probability = E(wait ^ (__ > n))
    return as_scalar( probability )
```



```
def wait_exactly(n, q=symbol('q')):
    "Returns probability of waiting more than n flips for heads; q is the weight on tails."
    wait = iterate(wait_for_heads, n + 2, constant(1), q=q)
    probability = E(wait ^ (__ == n))
    return as_scalar(probability)
```

Try calling each of these for a few values, like `wait_more_than(4)`, `wait_more_than(10)`, `wait_exactly(3)`, `wait_exactly(7)` and so forth. You will see that for any n `wait_more_than(n)` returns q^n and `wait_exactly(n)` returns $q^{n-1}(1 - q)$. The latter makes immediate intuitive sense: to first get a heads on the n th flip, we need to get $n - 1$ tails followed by a heads. And because we have an independent mixture of flips, the probabilities multiply.

When we iterate to find the expectation of the waiting time, it will necessarily be approximate because in principle one can wait an arbitrarily long time to see the first heads. However, the approximation will be very good for reasonable heads because as we've seen, the probability of waiting longer than n flips is q^n which decreases very quickly. For example, `E(iterate(wait_for_heads, 11, constant(1)))` gives

$$1 + q + q^2 + q^3 + q^4 + q^5 + q^6 + q^7 + q^8 + q^9 + q^{10} + q^{11}$$

which is close to $1/(1 - q)$. Indeed, trying it for a few values of n shows the same form suggesting that the exact expectation is $\sum_{j=0}^{\infty} q^j = 1/(1 - q)$, which is 1 over the probability of heads. So if heads come up with probability $1/2$, we expect to wait 2 flips for a heads, if heads has probability $1/1000$, we expect to wait 1000 flips, and so on.

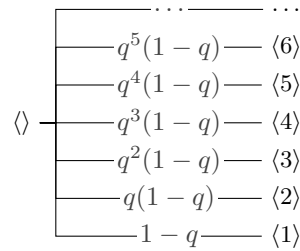
We can find this exactly by solving for the fixed point, the Kind `remaining_flips` that is unchanged by applying `wait_for_heads`. Hence, `remaining_flips` is equal to `wait_for_heads(remaining_flips)`. If we write down the Kind and equate the weights for branches of the same value, we can solve for all the weights. The fact that the Kind has an infinite number of leaves does not cause a problem.

Figures 45 and 46 show the process. In the first figure: on the left, we specify arbitrary weights for every possible number of flips, and on the right we apply the conditioning operator `wait_for_heads` using the `flips` Kind. The second

figure shows the same comparison, reducing the second tree to canonical form. Equating weights for each branch, we get

$$\begin{aligned} p_1 &= 1 - q \\ p_2 &= qp_1 = q(1 - q) \\ p_3 &= qp_2 = q^2(1 - q) \\ &\vdots \quad \vdots \end{aligned}$$

That is, $p_j = q^{j-1}(1 - q)$ for every integer $j \geq 1$, and the Kind we seek is



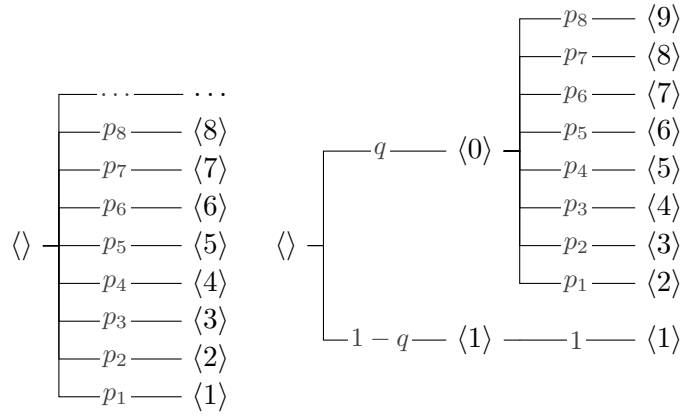
which does indeed have expectation $1/(1 - q)$.

We can take this idea and run with it. The next two examples use the same approach in slightly different situations.

Example 8.13 Double Heads and Other Patterns

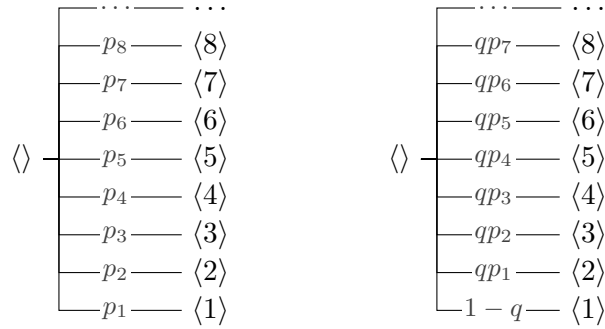
What if in the previous example we were not waiting for a single heads to come up but instead waiting for the first appearance of *two consecutive heads*? The approach would be the same, except we have to account for more than two states. In particular, we want to find the sequences that lead us back into the same state we were in at the beginning. That will enable us to solve the equation for the Kind of the total number of flips.

When waiting for two heads, there are *three* possibilities that lead us either to success or back to the state we started in. If we get a heads-heads, then we are done having used two flips. If get a heads-tails, we are back to our original state having used two flips. If we get a tails, we are back to our original state using one flip. Calling these possibilities 0 (tails), 2 (heads-tails), and 3 (heads-heads),



$$\text{remaining_flips} == \text{wait_for_heads}(\text{remaining_flips})$$

FIGURE 45. Solving for `remaining_flips`. On the left is the Kind `remaining_flips` with arbitrary weights assigned to each value. We want to solve for those weights in terms of q . On the right is the value of `wait_for_heads(remaining_flips)` which operates by conditioning on a single flip. (See that function earlier.)



$$\text{remaining_flips} == \text{wait_for_heads}(\text{remaining_flips})$$

FIGURE 46. Solving for `remaining_flips` continued. Here, we reduce the right-hand Kind in Figure 46 to canonical form. Making the two Kinds here equal just requires equating the waits on the branches for each value. So, $p_1 = 1 - q$, $p_2 = qp_1$, $p_3 = qp_2$, and so on. We can solve these equations for the p_i 's as shown in the text.

the conditional Kind describing this is

```
conditional_kind({
    0: remaining_flips ^ (__ + 1),
    2: remaining_flips ^ (__ + 2),
    3: constant(2)
})
```

which by direct analogy with `wait_for_heads` earlier gives us

```
def wait_for_2heads(remaining_flips, q=symbol('q')):
    """Returns the kind of the total number of flips to get consecutive heads.

    `remaining_flips` is the kind of the additional number of flips
        required after seeing a tails or heads-tails.

    `q` is the probability of getting a tails. [Default: symbol('q')]

    """
    # Make sure q is a symbol or high-precision number
    q = as_quantity(q)

    # The kind of the current prefix flips
    prefix = weighted_as(0, 2, 3, weights=[q, q*(1 - q), (1 - q) * (1 - q)])

    # the kind of the total number of flips given the current flip
    flips_given_current = conditional_kind({
        0: remaining_flips ^ (__ + 1),
        2: remaining_flips ^ (__ + 2),
        3: constant(2)
    })

    total_flips = flips_given_current // prefix
    return total_flips
```

Again, we can compute the probability of waiting more than n flips or of waiting exactly n flips. These are

```
E( iterate(wait_for_2heads, n + 1, constant(2)) ^ (__ > n) )
E( iterate(wait_for_2heads, n + 2, constant(2)) ^ (__ == n) )
```

just like before.

Solving the analogous “Kind equation” as before gives us the following.

$$\begin{array}{c}
 \langle \rangle - \begin{array}{l} \dots \dots \\ p_8 \text{---} \langle 8 \rangle \\ p_7 \text{---} \langle 7 \rangle \\ p_6 \text{---} \langle 6 \rangle \\ p_5 \text{---} \langle 5 \rangle \\ p_4 \text{---} \langle 4 \rangle \\ p_3 \text{---} \langle 3 \rangle \\ p_2 \text{---} \langle 2 \rangle \end{array} \\
 \mathbf{k}
 \end{array}
 \quad == \quad
 \begin{array}{c}
 \dots \dots \\
 qp_7 + (1-q)qp_6 \text{---} \langle 8 \rangle \\
 qp_6 + (1-q)qp_5 \text{---} \langle 7 \rangle \\
 qp_5 + (1-q)qp_4 \text{---} \langle 6 \rangle \\
 qp_4 + (1-q)qp_3 \text{---} \langle 5 \rangle \\
 qp_3 + (1-q)qp_2 \text{---} \langle 4 \rangle \\
 qp_2 \text{---} \langle 3 \rangle \\
 (1-q)^2 \text{---} \langle 2 \rangle \\
 \mathbf{wait_for_2heads(k)}
 \end{array}$$

Notice that $p_1 = 0$ has been excluded here as that is not a possible number of flips to get two heads.

Equating weights in these Kinds gives us a recurrence relation that we can solve:

$$\begin{aligned}
 p_2 &= (1-q)^2 \\
 p_3 &= q(1-q)^2 \\
 p_4 &= q^2(1-q)^2 + q(1-q)^3 \\
 &\vdots
 \end{aligned}$$

for all integers $j \geq 2$.

The same idea carries over to other patterns too. For some patterns, like consecutive strings of heads, solving for the Kind reduces a single equation between the Kind. In some cases, like heads-tails-heads-tails, we can express this as a system of “Kind equations.” We will see a general solution in an example in a later chapter.

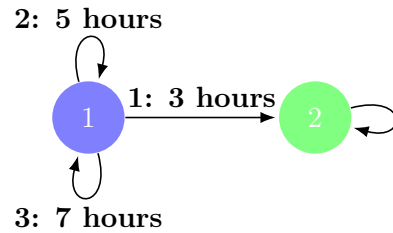


FIGURE 47. Graph describing the state machine for the rover.

Example 8.14 Recursive Rover

A robot exploring the Valles Marineris canyon system on Mars finds itself at the end of a canyon with three ancient river channels leading away. If it takes the first channel, it will reach its base site after 3 hours of rocky travel. If it takes the second or third channel, it will travel for 5 or 7 hours respectively only to find itself back where it started. If the robot (not a very bright one) always chooses among the channels randomly with equal weights. How long do we predict the rover will take until it reaches safety?

We can think of this as a machine with two states. In state 1, the robot is at the end of the canyon, and in state 2, it has returned successfully to base. From state 1, the robot can take any of the three channels, two return the machine to state 1, one to state 2. These transitions are shown in Figure 47, which depicts a graph commonly known as a Finite-State Machine. Each transition from state 1 is labeled with the time the robot requires to make that transition.

Let T and C be FRPs. T represents the # of hours until the robot reaches base, and C 's represents the channel (1, 2, or 3) that the robot chooses *initially*. We want to find $\mathbb{E}(T)$.

We will use the same technique as in the previous two examples. The function `time_to_base` finds the Kind of the robot's total time to base in terms of the Kind of the *remaining time* after the first choice.

```
def time_to_base(t):
    """Returns the conditional kind of time to base.
    Here, t is the *kind* of the remaining time *after the step*.
```

```

"""
base = conditional_kind({1: constant(3),
                        2: t ^ (__ + 5),
                        3: t ^ (__ + 7)})
channel = uniform(1, 2, 3)

return base // channel

```

And with this, can approximate $\mathbb{E}(T)$ quite well by iterating on a guess as before:

```

E(iterate(time_to_base, 10, constant(3)))  #== 14.792
E(iterate(time_to_base, 20, constant(3)))  #== 14.996
E(iterate(time_to_base, 30, constant(3)))  #== 14.9999

```

By increasing the number of iterations, we allow for the possibilities of longer sequences by the robot. The expectations converge rather quickly to 15.

We suspect that $\mathbb{E}(T) = 15$. Can we solve this exactly? Yes as before, but here things are even simpler because we only need the expectation. In particular, the Kind of the robot's time to base is the (infinite) Kind \mathfrak{t} such that \mathfrak{t} is *equal to* `time_to_base(\mathfrak{t})`, which necessarily means that $\mathbb{E}(\mathfrak{t}) == \mathbb{E}(\text{time_to_base}(\mathfrak{t}))$.

But `time_to_base` just does a conditioning operation, conditioning on a `channel` Kind. And $\mathbb{E}(\text{base} // \text{channel})$ is the same as $\mathbb{E}(\text{base}) // \text{channel}$, where $\mathbb{E}(\text{base})$ is a function of the initial channel, which we can view as a conditional Kind that returns a constant Kind for each input. Hence,

$$\begin{aligned}
 \mathbb{E}(t) &= \mathbb{E}(\text{time_to_base}(t)) \\
 &= \frac{1}{3} \mathbb{E}(\text{constant}(3)) + \frac{1}{3} (\mathbb{E}(t) + 5) + \frac{1}{3} (\mathbb{E}(t) + 7) \\
 &= 1 + \frac{2}{3} \mathbb{E}(t) + 4 \\
 &= 5 + \frac{2}{3} \mathbb{E}(t).
 \end{aligned}$$

Solving our equation $\mathbb{E}(t) = 5 + \frac{2}{3} \mathbb{E}(t)$ gives us $\mathbb{E}(t) = 15$ as expected.

Note that $\mathbb{E}(\mathfrak{t} \wedge (_ + 5)) = \mathbb{E}(\mathfrak{t}) + 5$ by the Additivity property in equation (7.12) and similarly with 7.

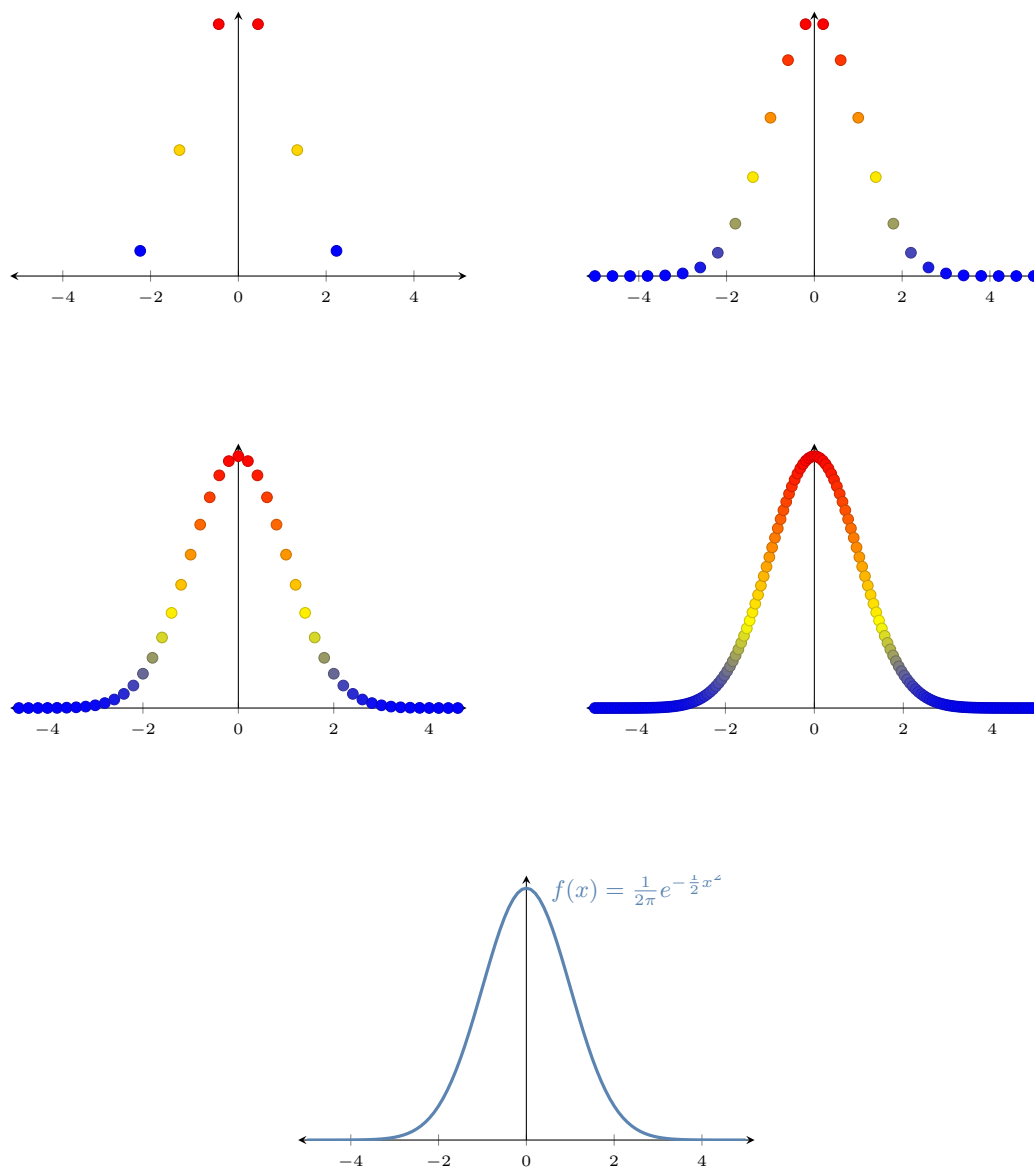


FIGURE 48. $\text{kind}(Z_n)$ for $n \in \{5, 25, 100, 1000\}$ in Example 8.15. Each Kind is shown by plotting the weights associated with each value, with one point per branch of the tree. The bottom panel plots the function $f(x)$ for comparison.

Example 8.15 Central Limit Theorem

For each $n \in [1..)$, let M_n be the FRP with Kind `Mean(either(-1, 1) ** n)`. (We can use `fast_mixture_pow` to compute this Kind as described earlier, see Puzzle 49.)

We will normalize these FRPs by scaling them in a particular way, defining

$$Z_n = \frac{1}{\sqrt{n}} M_n.$$

Figure 48 depicts the Kinds of Z_n for $n \in \{5, 25, 100, 1000\}$. Rather than showing the trees, we show the Kinds more compactly, plotting points $\langle v, w \rangle$ where v is the value of a branch and w is the associated weight.

All of these plots look like the function

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$$

up to a constant factor, and this function is shown in the bottom panel of the Figure for comparison. For even moderate n , the plotted points lie very close to this curve. The Kinds $\text{kind}(Z_n)$ have size growing with n , but the weights are simply described by this curve.

This is a special case of an important result called the **Central Limit Theorem**. The Kind of the normalized means Z_n converges in some sense in a way that lets us approximate the Kind with high accuracy.

Checkpoints

After reading this section you should be able to:

- Describe Finite Random Processes using mixtures, statistics, and conditionals.
- Formulate a strategy for analyzing Finite Random Processes with FRPs and Kinds.
- Use the Big 3+1 operations to compute Kinds and predictions.

Index

FRP

- dimension, [5](#)
- size, [5](#)
- type, [5](#)
- width, [5](#)

arbitrage price, [273](#), [280](#)

associativity, [134](#)

Bayes's Rule, [198](#), [223](#)

Big 3+1, [31](#)

bijection, [301](#)

bit, [291](#), [314](#)

Boolean expressions, [204](#)

canonical form, [116](#)

Central Limit Theorem, [375](#)

combinator

- statistic, [79](#), [80](#), [96](#), [99](#), [101](#), [181](#)

condition, [197](#), [203](#)

conditional, [72](#), [197](#), [200](#)

- constraints, [202](#)

- operator, [202](#)

conditional FRP

- mixture, [159](#)

conditional constraint, [25](#), [197](#), [205](#),
[205](#)

conditional FRP, [122](#), [151](#)

- codimension, [154](#)

- compatibility, [158](#)

- dimension, [154](#)

- domain, [154](#)

- target, [154](#)

- type, [154](#)

conditional FRP, FRP

- conditional, [154](#)

conditional Kind, [122](#), [157](#)

- codimension, [157](#)

- compatibility, [158](#)

- dimension, [157](#)

- domain, [157](#)

- mixture, [164](#)

- target, [157](#)

- type, [157](#)

conditional kind, [155](#)

conditioning, [123](#), [189](#), [192](#)

- operator, [189](#)

data-question, [189](#)

decorator, [49](#), [56](#)

determining class, [318](#)

dilation, [88](#)

dimension, [5](#)

entropy, [291](#), [313](#)

erosion, [88](#)

essential certainty, [276](#)

event, [67](#), [197](#), [203](#)

- complementary, [203](#)

- complements, [322](#)

- occur, [67](#)

expectation, [23](#), [33](#), [274](#), [286](#), [287](#)

factory, [162](#)

- FRP, [63](#), [85](#), [132](#)

- kind, [73](#), [132](#), [137](#), [256](#), [269](#)

- statistic, [67](#), [90](#), [95](#), [96](#), [98](#), [99](#), [303](#), [351](#)
- fixed point, [267](#)
- forest, [79](#)
- FRP given a condition, [205](#)
- frplib, [7](#)
 - combinators, [8](#)
 - factory, [8](#)
- FRP, [1](#)
 - clone, [131](#)
 - compatible, [46](#)
 - components, [62](#), [62](#)
 - dimension, [3](#)
 - factory, [63](#), [85](#), [132](#)
 - fresh, [1](#)
 - marginal, [62](#)
 - numeric, [2](#)
 - scalar, [3](#), [5](#)
 - transformed, [47](#), [47](#)
- function
 - anonymous, [74](#)
- graph, [63](#), [250](#)
 - edge, [63](#), [250](#)
 - node, [63](#), [250](#)
 - simple, [250](#)
 - undirected, [250](#)
 - without loops, [250](#)
- increment, [17](#)
- independent, [138](#)
- independent mixture, [22](#), [122](#), [128](#)
- independent mixture power, [122](#)
- indicator, [204](#)
- indicators, [50](#), [56](#)

- Iverson braces, [50](#), [204](#), [204](#), [322](#)
- Kind, [1](#), [3](#)
 - canonical form, [54](#), [108](#), [114](#), [116](#)
 - conditional, [155](#), [157](#), [225](#)
 - constant, [11](#)
 - dimension, [5](#)
 - equivalence, [108](#), [112](#)
 - factory, [73](#), [132](#), [137](#), [256](#), [269](#)
 - given condition, [206](#)
 - marginal, [62](#)
 - mixture
 - independent, [136](#)
 - size, [5](#)
 - transformed, [51](#)
 - type, [5](#)
 - weights, [4](#)
 - width, [5](#)
- linearity, [286](#)
- marginal, [188](#)
- marginalization, [62](#)
- Markov property, [249](#)
- Mean Absolute Deviation, [290](#)
- mixture, [122](#), [125](#), [148](#), [159](#), [164](#)
 - independent, [33](#), [126](#), [130](#), [287](#)
 - clone construction, [129](#)
 - flat construction, [129](#)
 - independet
 - flat method, [129](#)
 - wiring diagram, [125](#)
- mixture-marginal, [189](#)
- model, [19](#), [21](#), [87](#)
- monoid, [142](#)

probability, [71](#), [275](#), [321](#)

projection, [57](#), [57](#)

relation

binary, [113](#)

equivalence, [113](#)

equivalence class, [113](#)

risk-neutral price, [15](#), [273](#), [277](#), [281](#)

scalar, [2](#)

selector switch, [148](#)

set

increment, [17](#)

size, [5](#)

state, [239](#)

statistic, [19](#), [23](#), [32](#), [35](#), [39](#), [45](#)

codimension, [35](#), [45](#)

combinator, [79](#), [80](#), [96](#), [99](#), [101](#),
[181](#)

condition, [67](#), [197](#)

dimension, [35](#), [45](#)

expression, [96](#), [98](#)

factory, [67](#), [90](#), [95](#), [96](#), [98](#), [99](#), [303](#),
[351](#)

inline, [47](#), [47](#), [204](#)

scalar, [45](#)

type, [35](#), [45](#)

substitution property, [306](#)

transition, [250](#)

tree, [68](#)

type, [5](#)

function, [325](#)

unit, [325](#)

uncertainty, [276](#)

variance, [275](#), [290](#), [295](#), [317](#)

shortcut, [295](#)

weight, [1](#)

width, [5](#)