Stat 218

Chapter 0: All of Probability in a Box

Contents

v1.1.1 Last Updated 30 Sep 2023 at 2350

1 Introduction

Our goal this semester is to develop powerful tools to model and analyze random systems so that in the face of uncertainty, we can make better decisions, devise better algorithms, solve challenging problems, and make effective use of data.

But human intuition is famously terrible at reasoning under uncertainty. Even simple problems can be quite slippery. So, we develop a mathematical framework – called *probability theory* – to express concepts and operations precisely and help us build intuition about them in a real-world context. Probability theory is a rich and technical subject, but its core ideas and mechanics are actually quite simple. The most common operation reduces to taking *weighted averages*.

In this chapter, we cover *all* the core ideas in probability theory using random systems that produces a *finite* collection of values. The calculations are straightforward, there is very little math, and we can model many random systems of interest even with this restriction. The good news is that the concepts and operations we use here carry the same meaning as in the general case we will cover later. The general case will expand the reach and power of our tools, at the cost of some complexity and abstraction. Here, we keep things simple while building a solid foundation of understanding.

Most of the technical complexity in probability theory stems from the challenge of handling infinite sets in a sensible way.

Learning how to *do* the calculations is helpful, but to use probability theory effectively, it is critical to understand what the underlying quantities *mean*, both conceptually and in the context of the real systems we are modeling. Meaning will guide you in deciding what variables to define, which calculations to do and when, and how to express the solution to a problem cleanly. Meaning also reveals the common structure in what can seem like different ideas, deepening understanding. Throughout these notes, we will capture that meaning through a series of models, stories that give a framework for interpreting the quantities and operations we use.

In this chapter, our model centers on hypothetical devices – called Fixed Random Payoff boxes, or FRPs for short – with a simple behavior: push a button, get a payoff. Although it is not obvious at first, the key question is: How much should we pay to push the button on an FRP? The answer to that question quantifies our ability to *predict* the payoff and thus forms the basis for all of our analysis in probability theory. With a few simple rules, we can analyze FRPs fully, and given a real-world random system, we will see how to model that system with an FRP.

2 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 a payoff that goes to the owner of the FRP, 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 allowing us to study each kind *in aggregate*. Examining the values of many FRPs of the same kind – across a variety of kinds – helps us understand what to expect from FRPs.

An FRP kind is a complete tree with a positive, numeric ***weight*** on each edge and a list of numbers at each node. The leaf nodes give the possible ***values*** that the FRP can produce; each value is a list of numbers. And a path from the root to the leaf shows a sequence of numbers being successively added to the list, which starts out empty at the root. The lengths of the lists at the leaf nodes must all be the same and 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*.

So if you own an FRP, you hold the promise of some value. Before you push the button, you do not know what that value will be, and once you've pushed the button,

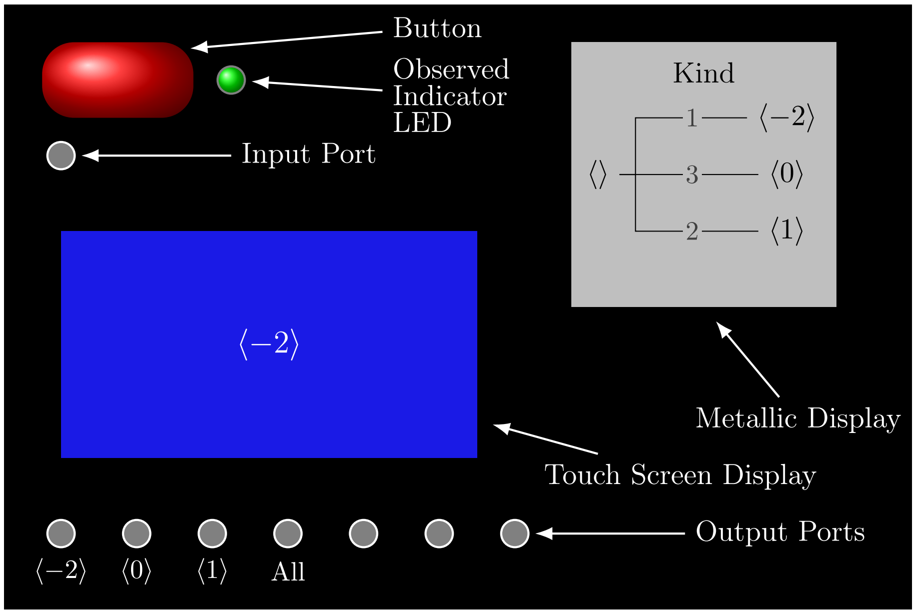


FIGURE 1. The top face of a typical fixed random payoff box, or FRP. The text will explain the role of each element shown here, including Kinds in section 2.1 and the input/output ports in Section 4 and 5.

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).

Here, we will only consider FRPs that output *lists of numbers*, and we will require that all the 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. 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. The *length* of a list is the number of elements it has. A list of length 1 is called a ***scalar*** and is treated specially. In practice, we make no distinction between the list with one element and the value 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*** n if it outputs only lists of length n. If it has dimension 1, we call it a ***scalar FRP***.

It 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.

We will formalize our treatment of lists, tuples, and vectors in the next chapter.

How do we interpret an FRP's (fixed for all time yet randomly produced) value?

As the P in FRP suggests, 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 its value $\langle v_1, v_2, \ldots, v_n \rangle$ as a menu of n payoffs, and we must choose how much of each we want *before* we see the value. But more on that later; for now concentrate on the scalar case.)

How much is an FRP worth? An answer to that question is a *prediction* about the FRPs value. A good answer incorporates all the information we have about the FRP at a given time, accounting for the uncertain outcome. Understanding how to answer that question is at the core of probability theory and will be our focus for the rest of this chapter (and then some).

Because each FRP produces just a single value in mysterious ways, we need more information if we are to make good predictions about the value. Fortunately, the metallic display on each FRP depicts its *kind*, and we will see that FRPs with the same kind behave similarly. So by considering a large collection of FRPs of the same kind, we can learn what we need to make good predictions.

2.1 Kinds

The ***kind*** of an FRP describes the nature of the FRP in a way that we will explore empirically. A kind is a *tree* where the values at the leaves indicate all the possible values that an FRP of that kind might output. All these values must be *distinct*, and they all must have the same type (in particular, the lists at each leaf must have the same number of elements). A tree representing a kind must also be *complete*, meaning that every path from root to leaf has the same length. Each edge in the tree has an associated number. These numbers are called the ***weights***. All the weights must be *positive numbers*. We will learn what the weight means as we play with FRPs below.

For reasons that will become clear later, we allow for the trivial tree with only a root node $\langle\rangle$ to represent an FRP that returns only an empty list.

Figure 2 shows a few simple examples of kinds and Figure 4 shows a more complicated example. 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 below. Figures 2 and 3 show the same kinds with horizontal and vertical layout to help you get used to the former.

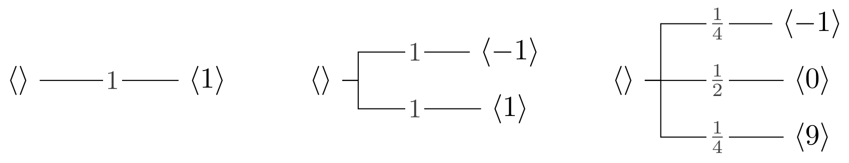


FIGURE 2. Several FRP kinds, all with dimension 1 (scalar) and with sizes 1, 2, and 3, respectively. The leaves, which must be distinct, give the possible values that can be output, and the positive numbers on the edges are called *weights*.

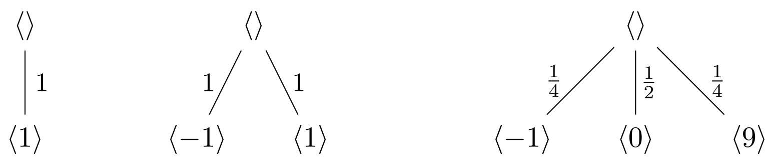


FIGURE 3. The same kinds as in Figure 2 but displayed with the root at the top. These more familiar orientations are intended to help you get used to the horizontal layout. You can use either orientation in your work, but these notes will use the horizontal layout in what follows.

We say that an FRP and its kind have ***size*** m if the kind has m leaves; that is, the FRP can produce one of m distinct possible values. We say that the FRP and its kind have ***dimension*** n if the list at every leaf of the kind has length n. FRPs of the kinds in Figure 2 have, respectively: size 1, 2, and 3. They all have dimension 1. The kind in Figure 4 has size 6 and dimension 2. The trivial FRP, called empty, has size 1 and dimension *0*; it's kind is just a root node with an empty list, denoted by $\langle\rangle$. We will look closely at these examples below. We call FRPs with dimension 1 ***scalar FRPs***.

For kinds of dimension bigger than 1, the structure of the tree gives us a picture of a random process at work in stages, generating numbers one at a time and collecting them into a list. Starting from the empty list at the root, the FRP generates and appends a number to the list at each level along a path from root to leaves, with the possible numbers determined at each stage by the path taken so far. The list 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. So, we can think of the FRP as generating numbers one at a time, contingent on the previous numbers

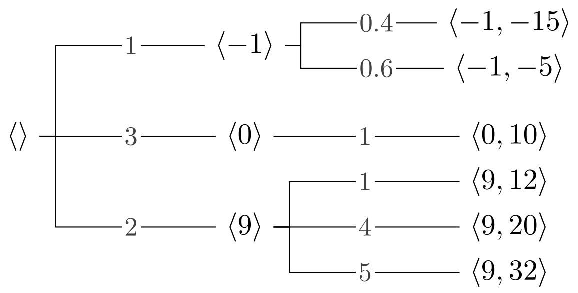


FIGURE 4. A more complicated FRP. We think of the values on the leaves as being generated in stages. Starting from an empty list at the root, the FRP first generates one of -1, 0, or 9 and appends it to the list, producing one of the lists shown at the first level. Then depending on whether -1, 0, or 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 that at each level the generated numbers must be distinct and the weights positive.

generated, and gathering these into a list. Or equivalently, we can think of the FRP as generating a list of numbers (from the set of values at the leaf nodes) all at once. Either way, the lists at the leaves show the distinct possible values we might see.

Puzzle 1. Draw the kind of an FRP of dimension 2 in which the second element generated is not influenced by the first element generated.

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.

2.2 FRPs at Scale

You have access to a warehouse containing a seemingly inexhaustible supply of FRPs. The warehouse manager does not like other people poking around in the warehouse, so you tell the manager the *kind* and number of FRPs you want, and the manager retrieves 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 highly automated, so you can manage the entire transaction with a program called `frp` that the warehouse makes available. With a single command from your computer, 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.

Many Bothans worked hard to bring you this information.

For a short time, you have a free trial where you can get as many FRPs as you like at no cost, and see their values. As this is a demo only, you receive no payoff when you push the button. We will use this free trial to 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.

Activity. Install the `frplib` package by following the instructions at `https://github.com/genovese/frplib`. This is a Python application and library that you can run standalone or use from your own programs. (There are several ways to invoke the program as described in the instructions on the GitHub page, including `python3 -m frplib`, but in what follows here, we will use the command `frp` as a placeholder.)

You run the `frp` application from the terminal command line on Mac OS, Linux, or Windows, invoking it with different *sub-commands* to use it in various ways, like `frp market` and `frp playground`.

In this section, we will focus on `frp market`. When you enter this at the terminal prompt, you will be repeatedly prompted for tasks to send to the FRP warehouse. When you see the prompt `market>` you can enter a task. These can span multiple lines and must end in a period (`.`). After the first line of a multi-line command, the prompt will become `...>` indicating an incomplete task. Type "`exit.`" or "`done.`" at the prompt to end your session, or "`help.`" for assistance.

Some of the market commands operate on a kind, and to specify that kind we use a simple text format that represents the tree. The program needs you to specify the kind of FRP to simulate. Details can be seen with the "`help kinds.`" task in the market. For example, the kind 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. Each value tuple is enclosed in `<>`. Whitespace, including any newlines, is ignored.

We will show the commands at the "`market>` " prompt and see the program's output in response. Any text from `#` to the end of a line is a comment for your benefit; you should not enter it when following along.

We can always use the `show` command to check that our input string gives the kind we expect. For example:

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

Now let's begin by examining the simplest, non-empty FRP, $\langle\rangle$ ——1—— $\langle 1\rangle$

```
market> demo 10000 with kind (<> 1 <1>).
Activated 10000 FRPs with kind (<> 1 <1>)
Summary of output values:
    1         10000 (100%)
```

(Notice that since this is a scalar FRP, the table elides the distinction between the value 1 and the value $\langle 1\rangle$.)

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 it can give. Try again with a different weight, for instance:

```
market> demo 10000 with kind (<> 0.001 <1>).
Activated 10000 FRPs with kind (<> 0.001 <1>)
Summary of output values:
  1         10000 (100%)
```

which yields the same result.

Try it yourself for a variety of weights. It seems that with an FRP whose kind has size 1, the weights have no influence, and it always outputs the value on the single leaf node.

So an FRP with kind $\langle\rangle \overset{}{\text{——}} w \text{——} \langle v\rangle$ is called a **constant FRP** with 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 ***constant kind*** without the (irrelevant) weight, which we will implicitly take to be 1: $\langle\rangle \text{————} \langle v\rangle$ is the constant v.

Next, consider the kind $\langle\rangle \left[\begin{array}{l} \text{—}1\text{——} \langle 0\rangle \\ \text{—}1\text{——} \langle 1\rangle \end{array}\right.$ 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:

```
market> demo 10_000 with kind (<> 1 <0> 1 <1>).
Activated 10000 FRPs with kind (<> 1 <0> 1 <1>)
Summary of output values:
  0         5031 (50.31%)
  1         4969 (49.69%)
```

(In the market, numbers can contain _ to separate blocks of three digits and make the numbers more readable.) The numbers of 0's and 1's are almost equal. Trying it with a larger number of FRPs might give

```
market> demo 1_000_000 with kind (<> 1 <0> 1 <1>).
Activated 1000000 FRPs with kind (<> 1 <0> 1 <1>)
Summary of output values:
  0       499895 (49.99%)
  1       500105 (50.01%)
```

This suggests a hypothesis about the weights. Let's vary the weights:

```
market> demo 1_000_000 with kind (<> 1 <0> 4 <1>).
Activated 1000000 FRPs with kind (<> 1 <0> 4 <1>)
Summary of output values:
  0       200300 (20.03%)
  1       799700 (79.97%)
```

So when the weights were both 1, we saw about the same number of 0's and 1's, but when the weights were 1 and 4, the relative frequencies of 0's and 1's produced was 1 to 4. Let's scale up the weights:

```
market> demo 1_000_000 with kind (<> 100 <0> 400 <1>).
Activated 1000000 FRPs with kind (<> 100 <0> 400 <1>)
Summary of output values:
  0       199987 (20.00%)
  1       800013 (80.00%)
```

and down:

```
market> demo 1_000_000 with kind (<> 0.2 <0> 0.8 <1>).
Activated 1000000 FRPs with kind (<> 0.2 <0> 0.8 <1>)
Summary of output values:
  0       199987 (20.00%)
  1       800013 (80.00%)
```

The frequencies are the same: 1 to 4.

Puzzle 3. If I give you a choice between two FRPs, with actual payoff, with kinds `(<> 10 <-1> 30 <0> 20 <10>)` and `(<> 100 <-1> 300 <0> 200 <10>)`, which do you prefer and why?

Back up your preferences with evidence from the `frp market`.

Puzzle 4. If you demo a large number of FRPs with kind `(<> a <0> b <1> c <2>)`, 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?

The `buy` command in the market allows you to purchase FRPs of specified kinds and numbers at specified prices. Since we are still in our free trial, there are no consequences, so it is like `demo` but computes our net payoffs:

```
market> 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
```

Puzzle 5. If you run a demo with kind `(<> 2 <-5> 3 <0> 5 <4>)` and specify the price per FRP (the number after the `@` in the `buy` command), how would you use the demo results and the price to compute a table similar to what the `buy` command shows.

Can you go the other way? What if the kind were of the form `(<> a <0> b <1>)` for some weights a and b?

Note: in any run of the `demo` and `buy` commands, the results will be slightly different because different FRPs are being activated, even if they have the same kind.

To test our intuition, let's consider a kind of size 2.

```
market> demo 1_000_000 with kind
...>      (<> 1 (<0> 1 <0, 0> 2 <0, 1> 3 <0, 2>)
...>          2 (<1> 1 <1,1> 1 <1, -1>)).
```

```
Activated 1000000 FRPs with kind
  (<> 1 (<0> 1 <0, 0> 2 <0, 1> 3 <0, 2>)
      2 (<1> 1 <1,1> 1 <1, -1>))
Summary of output values:
<0, 0>      55450 ( 5.54%)
<0, 1>     111469 (11.15%)
<0, 2>     166595 (16.66%)
<1, -1>    333562 (33.35%)
<1, 1>     332924 (33.29%)
```

These results are a bit more mysterious, and we will see exactly where they came from in the next section. But for now, look closely at the weights and think about the process in stages. Try varying the weights slightly in the market. For instance, what happens with weights `(<0> 1 <0, 0> 1 <0, 1> 3 <0, 2>)` on the top subtree of the kind? Putting this together, can you get a sense of why the demo shows these relative frequencies?

Puzzle 6. Considering your explorations over the last few puzzles and examples, how would you summarize your findings in a sentence? Can you state a general hypothesis about what the weights tell us?

Explore with further demos to see what you can learn about what the kind of an FRP tells us about its output.

Activity. Use the `frp` program to empirically explore the meaning of FRPs and their kinds. Start with kinds of dimension 1 and convince yourself of the above hypothesis. What then can you deduce about higher dimensional kinds? Try some simple cases to connect the weights on the tree to the proportions in the summary output.

Puzzle 7. We have seen empirically that an FRP with kind `(<> 1 <0> 1 <1>)` will produce roughly even proportion of 0's and 1's. How does this depend on the number of FRPs you sample?

To investigate this, start with a couple demos of 100 FRPs of this kind. How close are the frequencies of 0's and 1's to 50%. How many FRPs do you have to include in your demo for these frequencies to get an extra decimal point closer to 50%?

2.3 Why We Care

We have gotten an introduction to what FRPs and their kinds are and gotten an inkling of how they behave. In the next few sections, we will develop these ideas fully so that we can make good predictions about FRPs output. These predictions – and the calculations underlying them – gives us probability theory in a nutshell. The big payoff comes in Sections 9 and 10 when we see how to use FRPs to model random processes in the real world and find that we already know how to answer all the questions we need. But as motivation toward that payoff, here is an example to give you a sense of what is to come.

The (in)famous Monty Hall game 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.

Once your strategy has been specified, we build a corresponding FRP to represent the decisions made at each stage of the game. Those choices 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 decisions are illustrated in Figure 5.

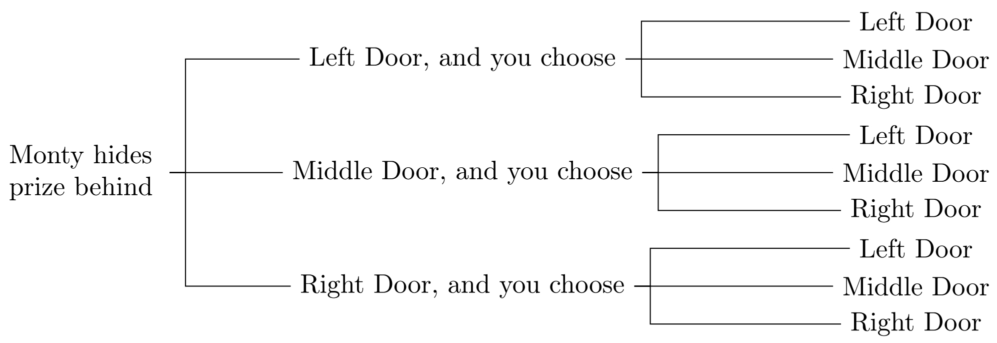


FIGURE 5. The decision tree leading to your initial door choice in the Monty Hall game. For any given strategy, these are the choices that determine the outcome of the game.

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 6. 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.

Puzzle 8. 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 of the game, 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.

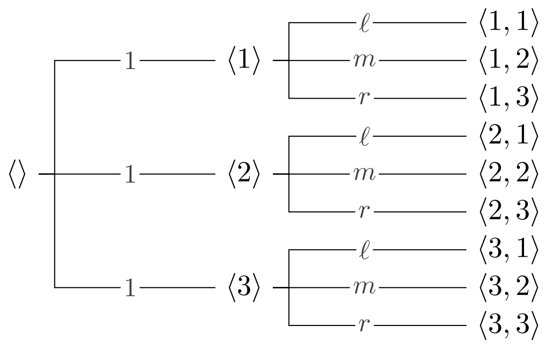


FIGURE 6. The kind for the FRPs modeling the Monty Hall game. In each value list, the first element is Monty's door choice and the second element is your door choice. The weights ℓ, m, and r are discussed in the text.

Puzzle 9. (Important!)

For each leaf node in Figure 6, 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 $d_{\text{Monty}} \neq d_{\text{You}}$.

As we will see, we can now transform the output of each FRP 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 7 for the no switch case (top) and the switch (bottom).

We will see how to do this in Section 5.

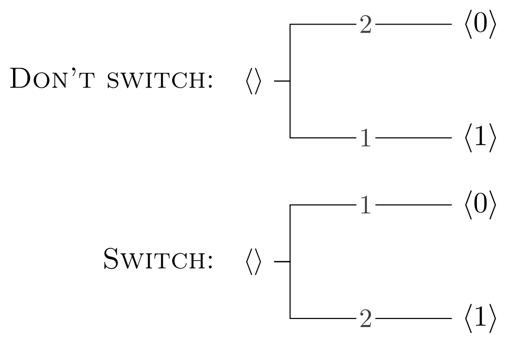


FIGURE 7. The kind for the transformed FRPs in the no-switch and switch cases, respectively. Notice that the weights in the two cases are different and that they do not depend at all on ℓ, m, or r.

Later we will use the `frp playground` command to build and play with FRPs and their kinds. Here is a preview related to this example.

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

We first load from `frplib.examples.monty_hall` two pre-defined kinds and a predefined "statistic":

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

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 all three doors.

We can combine these with an independent mixture to get the kind in Figure 6. (What is the playground expression for that mixture?)

We then *transform* that kind into new kinds using statistics. A statistic takes the value produced by an FRP as input and computes a new value, possibly of different dimension, as output. For example,

Don't worry about the details here; we will cover it soon. See the "Playground Overview" on page 36 and the `frplib` Cheatsheet for a summary.

```
                              got_prize_door_initially
```

extracts from the game outcome a 1 when your initial door choice hides the prize, or 0 if not. From that, we define the complementary statistic `didnt_get_prize_door_initially`

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

We use these statistics to produce the kinds of the game results under both the Don't Switch and Switch strategies:

```
playground> game_outcome = door_with_prize * chosen_door  # Kind in Fig 6

playground> dont_switch_win = got_prize_door_initially(game_outcome)
playground> switch_win = didnt_get_prize_door_initially(game_outcome)

playground> dont_switch_win
    ,---- 2/3 ---- 0
<> -+
    `---- 1/3 ---- 1

playground> switch_win
    ,---- 1/3 ---- 0
<> -+
    `---- 2/3 ---- 1
```

Finally, we activate FRPs of each kind to see what we should do in the game.

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

playground> 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`, respectively. The results are clear cut: switching seems like the right choice.

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*.
- Use the `frp market` to examine the values of FRPs of a given kind.
- Make some educated guesses, based on your work in the market, about an FRPs value from its kind.
- Explain roughly how FRPs might be useful for modeling real systems.

3 Equivalent Kinds

Key Take Aways

Kinds are described by weighted, complete trees, yet as we intuited in our explorations earlier, different trees can be *equivalent* in terms of predicting an FRP's value.

Two kinds k and k' with the same size, dimension, and set of values on their leaves are equivalent if given a collection containing any number of FRPs which each have either kind k or kind k', one cannot distinguish the kinds of the FRPs using their values, even in the aggregate. We can define this formally in terms of prices. 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 by Algorithm COMPACT.

Every kind tree has a ***canonical*** form, which we can obtain (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. The calculations that carry out this transformation will arise later in our prediction methods.

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, in that as we demo more and more FRPs of that kind, the relative frequencies of the values we see in the demo 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. Here, we take a closer look at when two kind trees are equivalent and see how to convert among different representatios of equivalent kinds. Such conversions will be useful when we calculate predictions in coming sections.

We will consider a game where an adversary offers you a batch of FRPs at a per unit price c, but instead of getting the value of each FRP for that price as a payoff, you get that value transformed by a rule that the adversary specifies in advance. The price may depend on the transformation rule, and the transformation rule must be defined on all possible values of the FRP. For instance, the adversary might choose to pay you \$1 if the FRP's value is positive and pay you -\$1 otherwise; or to pay you the square root of the FRPs value.

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 1. Two kinds k and k' are ***equivalent*** if both the following conditions hold

1. k and k' have the same size, dimension, and set of values on their leaves.
2. For any transformation rule that our adversary specifies, we are indifferent between purchasing m transformed FRPs of kind k at price c and purchasing any mixture of FRPs of kinds k and k' at the ***same price***.

Here, m can be any positive integer and c any real number. Recall that purchased

Negative prices mean we are *paid* to acquire the FRP.

FRPs are fresh, so their buttons have never been pushed when we check this condition.

Another way to state this definition is that if you had a collection containing any number of FRPs which each of which is either kind k or kind k', you cannot distinguish the kinds of the FRPs using their values, even in the aggregate. We could not, for instance, separate the FRPs into groups by kind, even after seeing the FRPs' values.

It is easy to check condition 1 of this definition by inspection, but it's less clear how to check condition 2. Condition 2 relates to the possible values and the corresponding weights along each path from root to leaf. The role of our adversary's transformation rule is to emphasize those paths in different ways, so if our predictions along any path are different between the two kinds, there will be a rule that would lead to different prices for the two kinds.

The good news is that given condition 1, condition 2 reduces to a few simple rules, making it easy to recognize equivalent kinds. We will determine those rules by considering the various arbitrary choices we make in specifying a kind.

One arbitrary choice we make in displaying a kind tree is the order of branches at each node. For example, in Figure 8, the following two kinds differ only in branch order and FRPs with these kinds would be indistinguishable in their behavior. *Kinds that differ only in the order of branches at a node are equivalent.*

We can choose a *canonical* branch order. 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 8 is in canonical branch order.

Another choice we have to make in specifying a kind is the size of the weights. This is not arbitrary like the branch order, but we do have some freedom. Consider the kinds in Figure 9. Are these distinguishable? You can try these in the `frp market` using the `compare` command, which acts like `demo` but takes *two* kinds with the same size dimension and value. For example,

```
compare 1000 with kinds (<> 1 <0> 1 <1>) (<> 0.5 <0> 0.5 <1>).
```

will print a table results by value for the two kinds. Can you tell these apart?

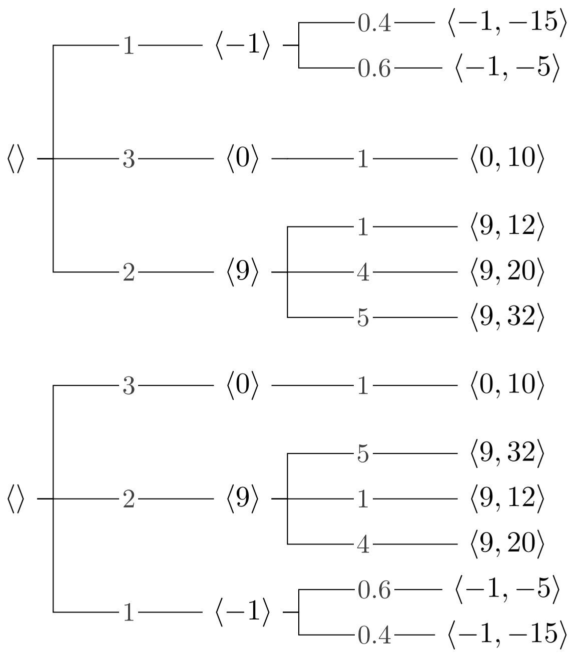


FIGURE 8. Two kinds that differ only in the order of branches at some nodes.

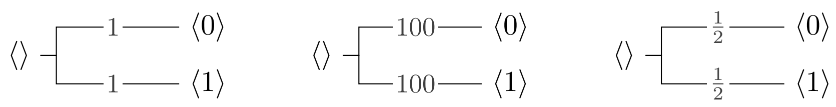


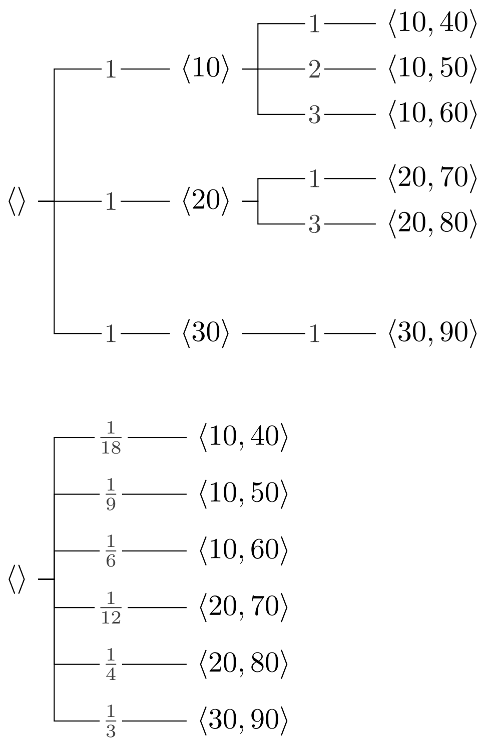
FIGURE 9. Three kinds whose weights differ only by a constant multiplicative factor.

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. *Kinds that differ only in a constant scaling of the weights weights branching from any node are equivalent.* This means that we have two kinds of the same size, dimension, and values and whose weights on the edges branching from some node, $w_1, \ldots, w_m$ and $w'_1, \ldots, w'_m$, satisfy: there is a $c > 0$ where $w_i/w'_i = c$ for every i.

We make a canonical choice of scalings: *the sum of the weights emanating from any node equals 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 number of levels to depict 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. But at the same time, there is redundant information in the multi-level presentation. Can you construct one kind in Figure 10 from the other? The kind shown on top is a multi-level tree; we call this the *unfolded* form. The kind shown on the bottom has the same size, dimension, and values but has one level; we call this the *compact* form. It might not seem obvious at first but the two are equivalent, and there are easy algorithms to go back and forth between them.

See Algorithms UNFOLDED and COMPACT.



$\langle\rangle$
1 — $\langle 10\rangle$
1 — $\langle 10, 40\rangle$
2 — $\langle 10, 50\rangle$
3 — $\langle 10, 60\rangle$
1 — $\langle 20\rangle$
1 — $\langle 20, 70\rangle$
3 — $\langle 20, 80\rangle$
1 — $\langle 30\rangle$
1 — $\langle 30, 90\rangle$
$\langle\rangle$
$\frac{1}{18}$ — $\langle 10, 40\rangle$
$\frac{1}{9}$ — $\langle 10, 50\rangle$
$\frac{1}{6}$ — $\langle 10, 60\rangle$
$\frac{1}{12}$ — $\langle 20, 70\rangle$
$\frac{1}{4}$ — $\langle 20, 80\rangle$
$\frac{1}{3}$ — $\langle 30, 90\rangle$

FIGURE 10. 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.

Every kind has a ***canonical form***, which we can find with 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.

And here is the key principle of equivalence.

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

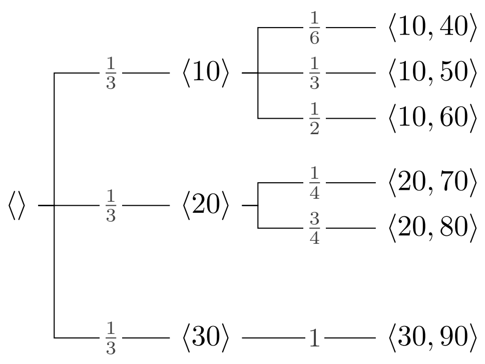
So, from here on, when we consider a kind, we will effectively identify it with the class of kinds that are equivalent to that tree. We treat the canonical form as the (typical) representative of this class, though we can freely translate among different equivalent trees when it gives us insight or helps with calculations.

At this point, it will be useful to establish a naming convention for FRPs and kinds, to make it easier to reference them. We will use capital Roman letters (like X, Y, and Z) to name FRPs, 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_3 and so forth. 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 $X_{[1]}, X_{[2]}, X_{[3]}, X_{[4]}$.

We will also use adorned letters like $k, k', k_1, \ldots$ to refer to particular but unspecified kinds.

If X is an FRP, $\mathsf{kind}(X)$ is its kind, $\mathsf{dimension}(X)$ is its dimension, $\mathsf{size}(X)$ is its size, and its set of values is $\mathsf{values}(X)$. Again, $\mathsf{kind}(X)$ really refers to an equivalence class of trees, though we can display it as any of the equivalent trees, 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 top of Figure 10. 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.

You use the `frp playground` command to view and manipulate kinds and understand this transformation. Within the playground, you can assign FRPs and kinds to variables. Pick a kind, apply the algorithms, and then use commands like the following to see both forms.

When 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.

```
playground> practice_1 = '(<> 10 (<3> 1 <3, 2> 7 <3,3>)
...>                           11 (<30> 4 <30,0> 8 <30,2>))'
playground> k1 = kind(practice_1)    # The kind specified by practice_1
playground> unfold(k1)               # shows the unfolded form
playground> 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, looking at the bottom tree Figure 10. 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 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 $\mathcal{L}_1, \ldots, \mathcal{L}_m$ (for some $m \ge 1$)
           of leaf nodes whose values are equal excluding the last element.

        ii. For $\mathcal{L}_j$ with $j \in [1 \mathinner{..} m]$, do the following:

            a. Let $n_1, \ldots, n_k$ be the leaf nodes in $\mathcal{L}_j$, with values of the form
               $\langle v_1, \ldots, v_{d-1}, x_i \rangle$ $d = \mathsf{dimension}(S)$ and $i \in [1 \mathinner{..} k]$, where $v_1, \ldots, v_d$
               are the same for all $k$ nodes and $x_1, \ldots, x_k$ are distinct.
            b. Modify $T$ by removing the edges from the common parent of the
               nodes $n_1, \ldots, n_k$ 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
               $n_1, \ldots, n_k$.
            c. Set the value for node $b_j$ to $\langle v_1, \ldots, v_{n-1} \rangle$.
            d. If $w_1, \ldots, w_k$ are the weights on the edges from $n_1, \ldots, n_k$ to their
               original parent in $T$, set the new weight on the branch from $b_j$ to
               each $n_i$ to $w_i/(w_1 + \cdots + w_k)$ and the weight on the branch from
               $b_j$ to its parent to be $w_1 + \cdots + w_k$.

        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., $b_1, \ldots, b_m$) added in step ii.

Step 4. Return $T$.
```

Activity. Generate several kinds in a mixture of unfolded and compact forms. Apply the algoritms to convert each to the other form. Use the `frp playground` command to view each kind in both forms and check your answers.

Checkpoints

After reading this section you should be able to:

- 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 Making New FRPs: Building with Mixtures

Key Take Aways

We can combine FRPs by connecting output ports to input ports in several ways to build new FRPs. The resulting FRPs capture important concepts: generating random outcomes contingent on earlier outcomes (mixtures), transforming the output of an FRP by some algorithm (statistics), and making predictions based on partial information (conditionals). The operations on FRPs lead to directly analogous operations on kinds, which will be fundamental in building and analyzing models for real systems. By combining mixtures, statistics, and conditionals we can model – and answer – most questions that arise in probability theory.

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.

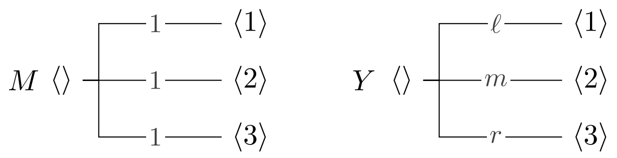
A ***mixture*** builds a higher-dimensional FRP from several lower-dimensional FRPs. One of these is called the mixer, and the rest are called targets, which must all have the same dimension. The mixture FRP selects one of the targets to activate *contingent* on the output value of the mixer. We hook the mixer's output ports to the targets' input ports, 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. First something happens, then something else happens depending on what happened initially, and so on. We start with the case of *independent mixtures*, where what happens at each stage does not influence what happens later. Then we generalize from there.

4.1 Independent Mixtures

Think back to the Monty Hall game in Section 2.3. 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) and Y for you, with respective kinds



We want to combine them to form a new FRP where the first part of the value (Monty's door) gets generated at the push of the button which triggers the second part of the value to be generated (your door) but in the same way regardless of the first part.

To do that, we can connect 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.
- When M's button is pushed, Y's display shows the combined value, and M's display is disabled.
- Y's output ports reconfigure (and relabel) automatically to give access to the combined value.

We call this FRP the ***independent mixture*** of M and Y and denote the resulting FRP by $M \star Y$. The kind $M \star Y$ is shown in Figures 11. The $\star$ operation on kinds means that we attach a copy of the $\mathsf{kind}(Y)$ tree to each leaf node in $\mathsf{kind}(M)$ and convert the new leaf nodes to hold the concanated lists, as shown in the Figure. Hence, we have the useful identity

$$\mathsf{kind}(M \star Y) = \mathsf{kind}(M) \star \mathsf{kind}(Y), \tag{4.1}$$

showing that FRPs and kinds combine in similar ways.

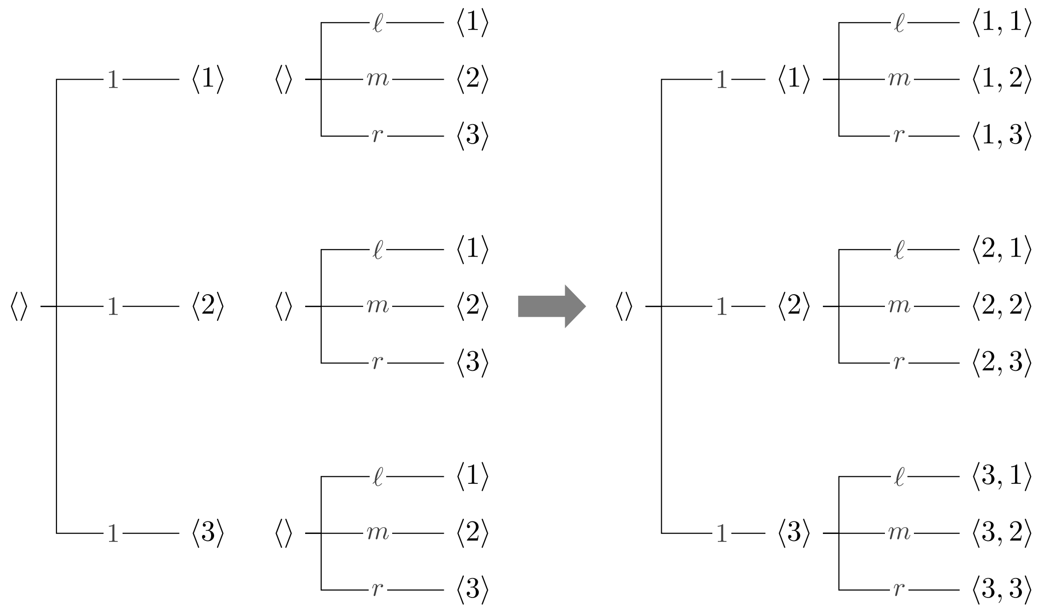


FIGURE 11. Constructing the kind mixture $\mathsf{kind}(M) \star \mathsf{kind}(Y)$. For each leaf of $\mathsf{kind}(M)$, we take a *copy* of $\mathsf{kind}(Y)$ and attach it, forming the kind tree on the right.

Puzzle 10. Convince yourself that equation 4.1 is true. Notice that after M generates a value, the next stage looks the same no matter what that value is.

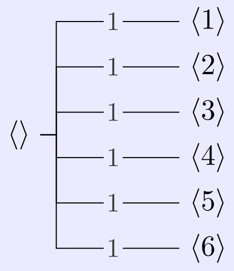
In $M \star Y$, we connected M to Y using the All port, but we can construct an *equivalent* FRP that looks more like Figure 11 by hooking *clones* of Y to the individual $\langle 1\rangle, \langle 2\rangle, \langle 3\rangle$ output ports of M. We do the following:

1. Obtain three "clone" FRPs $Y_{[1]}, Y_{[2]}, Y_{[3]}$ with kind equivalent to $\mathsf{kind}(Y)$.
2. For each $v \in \{1, 2, 3\}$, attach M's $\langle v\rangle$-output port to $Y_{[v]}$'s input port.
3. Push the button on M.

Connecting to the input ports disables the buttons on $Y_{[1]}, Y_{[2]}, Y_{[3]}$ and propagates the values similarly to the way described above, but: when M generates a value 1, it will trigger only $Y_{[1]}$; when M generates a value 2, it will trigger only $Y_{[2]}$; and when M generates a value 3, it will trigger only $Y_{[3]}$. The activated FRP will show the combined value on its display.

Puzzle 11. Convince yourself that the "All" method and the "clone" method of creating $M \star Y$ have the same kind.

Puzzle 12. In the classic game *Dungeons & Dragons*, each player has a character with various attributes (strength, charisma, etc.). Each attribute is determined by the rolls of three six-side dice. Let D be the FRP with kind



Express the roll of three dice as an independent mixture in terms of D. Sketch its kind. How does that kind relate to the kind shown just above?

One problem though: D&D players' character attributes are the *sum* of the three dice. How do we get that from the mixture you devised above? The answer is that we use a *statistic*, as described in Section 5 and shown in Example 4.1 below.

Puzzle 13. If X and Y are FRPs, is $X \star Y$ the same as $Y \star X$? If so, how do you know? If not, how are they related?

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 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, 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 $\mathsf{size}(X)$ *copies* of empty), the result is again the same as X. So, $X \star \mathsf{empty}$ is the same as X. That is:

$$X \star \mathsf{empty} = X = \mathsf{empty} \star X.$$

This applies to the kinds as well:

$$\mathsf{kind}(X) \star \langle\rangle = \mathsf{kind}(X) = \langle\rangle \star \mathsf{kind}(X).$$

For FRPs and kinds, respectively, empty and $\langle\rangle$ are "identity elements" for independent mixture.

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:

$$X \star (Y \star Z) = (X \star Y) \star Z$$
$$\mathsf{kind}(X) \star (\mathsf{kind}(Y) \star \mathsf{kind}(Z)) = (\mathsf{kind}(X) \star \mathsf{kind}(Y)) \star \mathsf{kind}(Z).$$

We say that $\star$ is "associative" for FRPs and kinds, and we can write the mixture without parentheses, $X \star Y \star Z$ and $\mathsf{kind}(X) \star \mathsf{kind}(Y) \star \mathsf{kind}(Z)$.

A set of objects (here either FRPs or kinds) with an associative binary operation and an identity element is called a *monoid*.

Example 4.1. Here we analyze the dice example above more fully. Invoke the `frp playground` command and follow along. First, we will generate the kind of D, then draw samples of FRPs with that kind. The `uniform` factory takes one or more values and produces a kind with equal weights on those values.

See "Playground Overview" on page 36 for guidance.

```
playground> d6 = uniform(1,2,3,4,5,6)  # All 6 rolls have equal weight
     ,---- 1/6 ---- 1
     |---- 1/6 ---- 2
     |---- 1/6 ---- 3
<> -|
    |---- 1/6 ---- 4
    |---- 1/6 ---- 5
    `---- 1/6 ---- 6
playground> FRP.sample( 12_000, d6 )   # Order 12,000 FRPs
Summary of output values:
 1         1959 (16.33%)
 2         1986 (16.55%)
 3         2096 (17.47%)
 4         2012 (16.77%)
 5         1977 (16.48%)
 6         1970 (16.42%)
```

Second, look at an independent mixture of two dice, $D \star D$. Examine the tree by entering `unfold(d6 * d6)`. (The `unfold` shows the full tree; if you enter `d6 * d6` instead, you will get an equivalent but more compact form, which we will discuss in the next section.)

Finally, try the case of three dice for your self. Compute the kind of $D \star D \star D$ and check its size and dimension and values with:

```
playground> three_d6 = d6 * d6 * d6
playground> three_d6.size
playground> three_d6.dim
playground> three_d6.values
```

I've excluded the output here, but you will see it. You can also look at the tree to compare it with your sketch from earlier. (It's large, though, so you may want to paste it into a file.)

As a preview of the next section, we can also compute the kind of an FRP that generates the sum of the three dice. Here we transform the tree using the `Sum` statistic into the kind the describes the sum of three dice.

```
playground> Sum(d6 * d6 * d6)
    ,---- 1/216  ---- 3
    |---- 3/216  ---- 4
    |---- 6/216  ---- 5
```

```
       |---- 10/216 ---- 6
       |---- 15/216 ---- 7
       |---- 21/216 ---- 8
       |---- 25/216 ---- 9
       |---- 27/216 ---- 10
   <> -|
       |---- 27/216 ---- 11
       |---- 25/216 ---- 12
       |---- 21/216 ---- 13
       |---- 15/216 ---- 14
       |---- 10/216 ---- 15
       |---- 6/216  ---- 16
       |---- 3/216  ---- 17
       `---- 1/216  ---- 18
```

This example highlights a common pattern, where we take an independent mixture of an FRP or kind with itself some number of times, like $D \star D \star D$. Because this is so common, we have a shorthand for it.

For any kind k and any natural number m, we define

$$k \star\star m = \overbrace{k \star \cdots \star k}^{m\ times}, \tag{4.2}$$

where $k \star\star 0 = \langle\rangle$.

For any FRP X and any natural number m, we define

$$X \star\star m = \overbrace{\mathsf{clone}(X) \star \cdots \star \mathsf{clone}(X)}^{m\ times}, \tag{4.3}$$

where $X \star\star 0 = \mathsf{empty}$. The clones here make the shorthand more useful; simply repeating the exact value of X is not what we usually want.

In the playground, we use the `**` operator for this, so these mixtures look like `X ** m` and `k ** m`.

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

`symmetric`, `linear`, `geometric` – weights on values determined by a symmetric/linear/geometric functions

`weighted_by` – weights on values determined by a general function

`evenly_spaced`, `integers`

`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 * 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 given a”) is equivalent to `a >> b ^ Project[-b.dim, -b.dim+1,...,-1]`

Playground Overview (cont'd)

Statistic Factories

`statistic`, `condition`, `scalar_statistics` – 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 the 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

`And`, `Or`, `Xor`, `Not` – logic operators

`Fork` - `Fork(f1,f2,...,fn)` applies each `fi` to its corresponding component

`ForEach` – apply a statistic to each component of a value

`IfThenElse` – if a condition is true, apply one statistic else another.

`^` – `s1 ^ s2` ("`s1` then `s2`")

`@` – `stat @ X` is like `stat(X)` but passes `X` to a following conditional

Actions

`symbol` – create a symbolic quantity with given name

`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

`bin` – group values of an FRP or kind into specified bins

`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

Help

`info` – frplib specific help

`help` – built-in python help

4.2 General Mixtures

The "clone" method for constructing an independent mixture generalizes immediately. By attaching *different* FRPs (of the same dimension) to each output of an FRP (e.g., M), we get an FRP where the second number generated is *contingent* on the first number generated. This gives us a general ***mixture***.

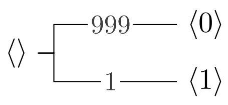
Example 4.2.

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 someone has the disease given that they test positive. Here, we will build an FRP to describe this situation.

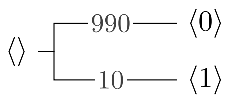
We will build an FRP by mixture in two stages. As before, we associate a number with each outcome:

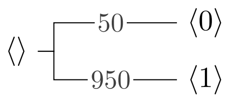
$0 \leftrightarrow$ No Disease	$0 \leftrightarrow$ Test Negative
$1 \leftrightarrow$ Disease	$1 \leftrightarrow$ Test Positive

The first stage corresponds to whether an individual has the disease, an FRP D with kind



The second stage corresponds to the test, and there are two possibilities depending on the outcome of the first stage. Call the FRPs N and P, respectively, with kinds:





Now, we attach the $\langle 0\rangle$ output port of D to the input port of N, and the $\langle 1\rangle$ output port of D to the input port of P. The resulting FRP is the mixture of D with N and P. More specifically, we define a mapping M from the values of D to the corresponding FRP, a rule that tells us to which input ports to connect each output port of D.

$$\mathsf{M}(\langle 0\rangle) = N$$
$$\mathsf{M}(\langle 1\rangle) = P.$$

We call this mapping a ***conditional FRP*** because it gives a rule for choosing an FRP conditionally based on a given value. We denote the resulting mixture FRP by $D \triangleright \mathsf{M}$ and call it the *mixture of* M *by* D.

Like a conditional in programming: "if we get a 0 then use N, else use P."

For kinds, things look analogous. We need a mapping m that associates the values of D to the corresponding *kinds* of N and P. Unsurprisingly, we call m a ***conditional kind***. We say that m is *compatible* with $\mathsf{kind}(D)$ if every possible value of $\mathsf{kind}(D)$ is a valid input to m. The *dimension* of m is the common dimension of the kinds it returns.

We need to ensure m is consistent with M in that $\mathsf{m}(0)$ should give the kind of $\mathsf{M}(0)$ and $\mathsf{m}(1)$ should give the kind of $\mathsf{M}(1)$. So it must be that:

$$\mathsf{m}(\langle 0\rangle) = \mathsf{kind}(N)$$
$$\mathsf{m}(\langle 1\rangle) = \mathsf{kind}(P).$$

Mathematically, we write $\mathsf{m} = \mathsf{kind} \circ \mathsf{M}$, which is read as "$\mathsf{m}$ after M." We denote the resulting mixture of kinds by $\mathsf{kind}(D) \triangleright \mathsf{m}$. This is illustrated in Figure 12.

You can play with Example 4.2 in the frp playground. Try the following to get started. 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.

```
playground> kindD = either(0, 1, 999)
playground> kindN = either(0, 1, 99)
playground> kindP = either(0, 1, 1/19)
```

First, we can define the conditional kind m in several ways: as a dictionary mapping values to kinds,

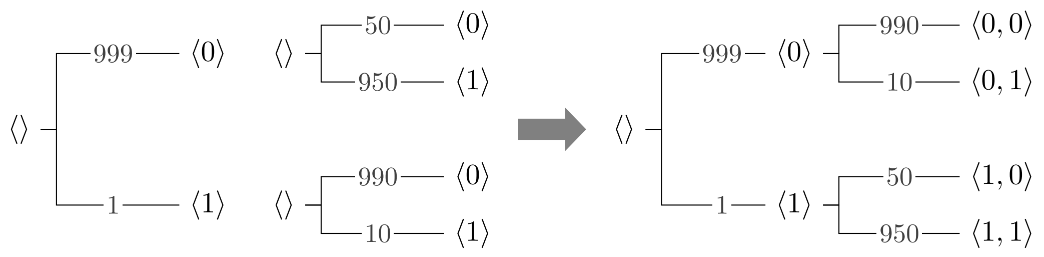


FIGURE 12. Constructing the kind mixture kind$(D) \triangleright$ m. For each value v of kind(D), we take the kind m(v) and attach it, forming the kind tree on the right.

```
playground> m = { (0,): kindN, (1,): kindP }
```

Here `(0,)` and `(1,)` are the tuples $\langle 0\rangle$ and $\langle 1\rangle$, with the extra comma distinguishing a tuple from a parenthesized number.

or as a (named) function,

```
playground> def m(value):
...>            if value == (0,):
...>                return kindN
...>            return kindP
```

or equivalently and more simply, as an anonymous function

```
playground> m = lambda value: kindN if value == (0,) else kindP
```

In Python, anonymous functions are introdued with the `lambda` keyword and have the form `lambda` *`args`*`:` *`expr`*, returning the value of expresssion *expr*.

Next, we can define the mixture of this conditional kind by `kindD`. In the playground, we use `>>` for the $\triangleright$ operator; see the **Playground Overview** on page 36.

```
playground> mixed = kindD >> m
playground> unfold(mixed)
```

Again, evaluating just `mixed` will show you a more compact tree.

In the above, we are using kinds, but the same operations work on FRPs:

```
playground> D = frp(kindD) >> { (0,): frp(kindN), (1,): frp(kindP) }
playground> D.value
<0, 0>
playground> FRP.sample( 1_000_000, D )
Summary of output values:
  <0,0>    988786 (98.88%)
```

```
    <0,1>     10176 ( 1.02%)
    <1,0>        44 ( 0.00%)
    <1,1>       994 ( 0.10%)
```

We are close to answering our original question here and will see a nice way to do it exactly. But we can see an approximate answer here; 994/10176 is the proportion of positive tests with the disease.

To define mixtures formally, it helps to look carefully at the pieces. For a mixture of FRPs, we need an FRP to generate the values in the first stage, the values we pass to downstream FRPs. We call this FRP the mixer. We need a collection of FRPs whose input ports we will connect the mixers outputs to; call these the targets. And finally we need a rule that tells us which targets to connect to which output ports of the mixer. This is a conditional FRP. And for kinds, we have analogous pieces.

Definition 2. Suppose R is an FRP of size s and dimension d_1 with possible values $v_1, \ldots, v_s$. Let $L_1, \ldots, L_s$ be FRPs of *equal dimension* d_2. We call R the mixer and $L_1, \ldots, L_s$ the targets.

A ***conditional FRP*** M is a mapping that associates each mixer *value* with a distinct target: $\mathsf{M}(v_i) = L_i$ for all $i \in [1\,..\,s]$. The dimension of M is the common dimension of the targets. Think of this as a rule that tells us how connect the output ports of the mixer with the targets.

An FRP R is compatible with a conditional FRP M when all the possible values of R are valid inputs to M.

Definition 3. Suppose k is an kind of size s and dimension d_1 with possible values $v_1, \ldots, v_s$. Let $k'_1, \ldots, k'_s$ be kinds of *equal dimension* d_2. We call k the mixer kind and $k'_1, \ldots, k'_s$ the target kinds.

A ***conditional kind*** m is a mapping that associates each mixer value with a target kind: $\mathsf{m}(v_i) = k'_i$ for all $i \in [1\,..\,s]$. The dimension of m is the common dimension of the target kinds.

An kind k is compatible with a conditional kind m when the values of k are valid inputs to m.

Definition 4. If k is a kind and m is a compatible conditional kind, we define the ***mixture*** $k \triangleright \mathsf{m}$ to be the kind with dimension $\mathsf{dimension}(k) + \mathsf{dimension}(\mathsf{m})$ obtained by attaching, for every value v of k, the target kind $(m)(v)$ to the v leaf node of k, and prepending v to every list in that subtree. This gives a larger tree.

If all the target kinds in m are equivalent to a kind k', then the mixture reduces to an *independent mixture*: $k \triangleright \mathsf{m} = k \star k'$.

Definition 5. If R is an FRP and M is a compatible conditional FRP, we define the ***mixture*** $R \triangleright \mathsf{M}$ is the FRP with dimension $\mathsf{dimension}(R) + \mathsf{dimension}(\mathsf{M})$ obtained by joining each output value v of R to the input of the corresponding target $\mathsf{M}(v)$.

We can also define a conditional kind that is compatible with $\mathsf{kind}(R)$ by $\mathsf{m}(v) = \mathsf{kind}(\mathsf{M}(v))$ for every value v of R.

The ***mixture*** $R \triangleright \mathsf{M}$ is the FRP with dimension $d_1 + d_2$ obtained by joining each output value of R to the input of the corresponding L_i.

The kind of $R \triangleright \mathsf{M}$ satisfies

$$\mathsf{kind}(R \triangleright \mathsf{M}) = \mathsf{kind}(R) \triangleright \mathsf{m}. \tag{4.4}$$

If, for all values of R, the targets in M are the *same* FRP L, then the mixture reduces to an *independent mixture*: $X \triangleright \mathsf{M} = X * L$.

The next example nicely illustrates the contingent evolution that mixtures capture.

Example 4.3. Theseus has awoken from a night of revelry to find himself trapped in a labyrinth ... again (Figure 13). 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

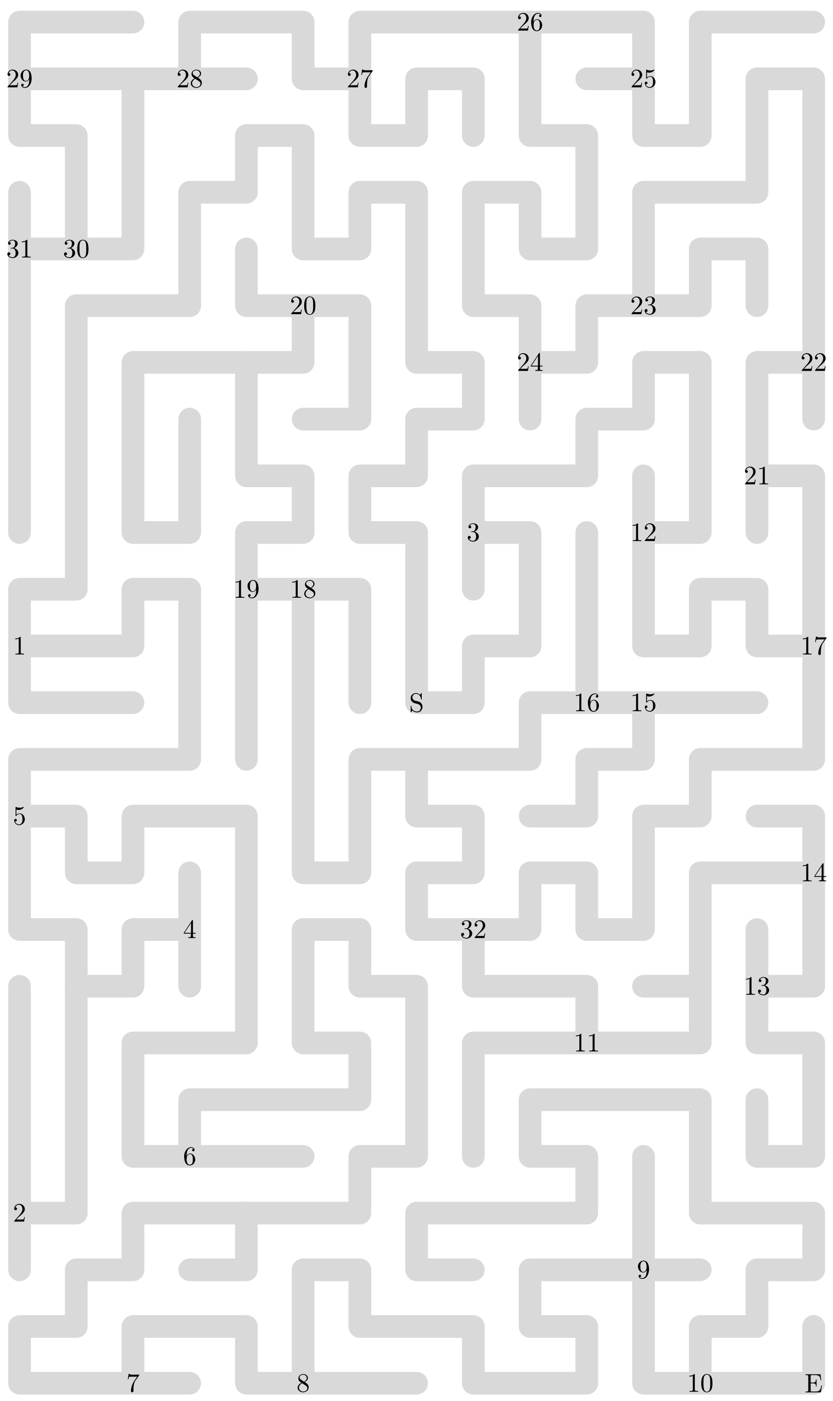


FIGURE 13. Another labyrinth that has ensnared poor Theseus. His starting point (S=0) and exit (E) are marked, and each juncture is assigned a number. The FRPs will generate a number corresponding to the juncture that Theseus wanders into.

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 by creating the kind of the FRP for Theseus's starting position. He begins at juncture 0 with certainty, so this is a constant.

```
playground> 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.

```
playground> from frplib.examples.labyrinth import *
playground> 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 but is not strictly necessary.

```
playground> steps = conditional_kind({
...>            juncture: uniform(neighbors)
...>            for juncture, neighbors in labyrinth.items()
...>        })
playground> steps
playground> moves = from_latest(steps)
```

In Python, this is called a *dictionary comprehension*.

These are the kinds of Theseus's moves at each stage, where `steps` gives the kinds for single step moves, and the call to `from_latest` ensures that `moves` can

`from_latest` also ensures that `moves` print out nicely.

handle a path of any length. Specifically, `steps` maps each juncture number to the kind for a move out of that juncture, and `moves` takes a *list* describing Theseus's path so far and uses `steps` to make a move based on the last juncture in the path.

Now, consider the FRPs that describe Theseus's path after one, two, and three moves. Take a moment to think about how we get the kinds of those FRPs from `start` and `moves`.

Puzzle 14. We could generate a table of FRPs at each juncture like

```
playground> 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?

At any juncture, `moves` gives the kind for a move *from* that juncture, so that will go on the right-hand side of `>>`. So this yields

```
playground> start                            # starting position
playground> start >> moves                   # after one move
playground> start >> moves >> moves          # after two moves
playground> 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
```

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 compact form; you can always use unfold to see the full tree, e.g., `unfold(start >> moves)`, though these can get big.

Puzzle 15. 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 borrow a trick from the next section 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`).

```
playground> move100_kind = after_move_n(100, start, steps)
playground> frp(move100_kind).value
playground> 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 16. 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.

Answers to Selected Puzzles.

Puzzle 14. 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 15. 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 16. 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 an independent mixture of FRPs or of kinds and what such mixtures mean.
- Describe what distinguishes an independent mixture from a more general mixture.
- Describe *conditional kinds* and *conditional FRPs* and show how to create them in the playground.
- Explain how to construct a general mixture between a kind and a conditional kind or between an FRP and a conditional FRP.
- Find the dimension, size, and values of $k \triangleright \mathsf{m}$ from the properties of k and m.
- Recover k from $k \triangleright \mathsf{m}$ by applying an appropriate statistic.
- Recover m from $k \triangleright \mathsf{m}$ by combining an appropriate conditional and statistic.
- Relate the kind of a mixture FRP X of an FRP and conditional FRP M to the mixture of the kind $\mathsf{kind}(X)$ and conditional kind $\mathsf{kind} \circ \mathsf{M}$.

5 Making New FRPs: Transforming with Statistics

Key Take Aways

We can combine FRPs by connecting output ports to input ports in several ways to build new FRPs. The resulting FRPs capture important concepts: generating random outcomes contingent on earlier outcomes (mixtures), transforming the output of an FRP by some algorithm (statistics), and making predictions based on partial information (conditionals). The operations on FRPs lead to directly analogous operations on kinds, which will be fundamental in building and analyzing models for real systems. By combining mixtures, statistics, and conditionals we can model – and answer – most questions that arise in probability theory.

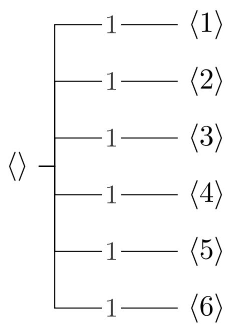
The data we collect from observing random processes is often high dimensional, but in most situations, we are interested in information derived from that data. A ***statistic*** is a data processing algorithm, a function that takes as input one of the possible values of an FRP and returns a value of a specified type. We transform an FRP by *applying a statistic to its output value*, yielding a new FRP with possibly different size and dimension.

The mixtures we saw in the last section showed the power of FRPs to captures systems that evolve in contingent ways. As the complexity of the systems grows, however, the FRPs produce more complex data. And in most cases, the questions we want to answer are about quantities *derived* from these data. For that, we use *statistics.*

Before we formalize this, let's look at a simple example of how we might apply a statistic to transform an FRP.

Example 5.1.

Consider a simple FRP D with kind



This might, for instance, be used to model the roll of a balanced 6-sided die. How would we capture the rolls of 5 such dice?

This is just an independent mixture: $D \star D \star D \star D \star D$. This is just the independent mixture of D with *itself* five times, so recall our shorthand for this from equation 4.3: $D \star\star 5$.

Each value of $D \star\star 5$ is a tuple of length 5, with each element a number from 1 to 6. The elements of the tuple give the values of successive "rolls of the dice." We use the values of $D \star\star 5$ as *data* to answer questions.

Most questions driving our analyses will tend to focus on information extracted from or summarizing this tuple, rather than the whole list. Consider some of the questions we might ask about the dice rolls:

1. What is the sum of the five rolls?
2. What is the value of the third die roll?
3. What is the maximum value of the five rolls?
4. How many rolls does it take until a 6 first appears, if at all?
5. How many times did the most common value appear among all rolls?
6. Did either pattern 1,2,3 or 4,5,6 ever occur on successive rolls?

We can answer each of these questions with *statistics*, which are functions/algorithms that process the tuple produced and give a different value to answer a question.

We use statistics to *transform* FRPs and kinds into new FRPs and kinds that answer more specific questions. In the frplib playground, we can transform an FRP or kind by applying the statistic as a function directly to the FRP or kind. It is sometimes more convenient to give the statistic separately; for this we use the `^` ("arrow") operator, with `an_frp_or_kind ^ statistic` equivalent to `statistic(an_frp_or_kind)`. The arrow operator is intended to be evocative of the "flow of data" from the FRP/kind and through the statistic.

For example, to answer the first three questions above, we could compute

```
playground> five_d6 = D ** 5
playground> Sum(five_d6)
An FRP of dimension 1 and size 26 with value 21
playground> five_d6 ^ Proj[3]
An FRP of dimension 1 and size 6 with value 4
playground> Max(five_d6)
An FRP of dimension 1 and size 6 with value 6
playground> kind(Sum(five_d6)).values
{5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
playground> Sum(kind(five_d6)) == kind(Sum(five_d6))
True
```

where `Sum`, `Proj`, and `Max` are pre-defined by `frplib`. `Sum` and `Max` compute the sum and the maximum, respectively, of the list of value produced by the FRP. `Proj[3]` is a *projection* statistic; it extracts the third element of the value; and `Proj[3](five_d6)` is equivalent to `five_d6 ^ Proj[3]`, which you choose is a matter of taste. (See the "Playground Overview" on page 36 for more.)

In the playground, you can also build statistics dynamically with expressions in terms of other statistics, numbers, and arithmetic operators. Here `__` acts as a placeholder for the value. For instance,

```
playground> bit = uniform(0,1)
playground> bit ^ (2 * __ - 1)
    ,---- 1/2 ---- -1
<> -|
    `---- 1/2 ---- 1
```

```
  playground> bit ** 5 ^ (Sum - 2.5)
      ,---- 1/32 ---- -2.5
      |---- 5/32 ---- -1.5
      |---- 5/16 ---- -0.5
  <> -|
      |---- 5/16 ---- 0.5
      |---- 5/32 ---- 1.5
      `---- 1/32 ---- 2.5
  playground> bit ** 3 ^ ForEach(2 * __ - 1)
      ,---- 1/8 ---- <-1, -1, -1>
      |---- 1/8 ---- <-1, -1, 1>
      |---- 1/8 ---- <-1, 1, -1>
      |---- 1/8 ---- <-1, 1, 1>
  <> -|
      |---- 1/8 ---- <1, -1, -1>
      |---- 1/8 ---- <1, -1, 1>
      |---- 1/8 ---- <1, 1, -1>
n      `---- 1/8 ---- <1, 1, 1>
  playground> uniform(1,2,3) ** 3 ^ ForEach(IfThenElse(__ % 2 == 0, 2, 3))
      ,---- 1/27 ---- <2, 2, 2>
      |---- 2/27 ---- <2, 2, 3>
      |---- 2/27 ---- <2, 3, 2>
      |---- 4/27 ---- <2, 3, 3>
  <> -|
      |---- 2/27 ---- <3, 2, 2>
      |---- 4/27 ---- <3, 2, 3>
      |---- 4/27 ---- <3, 3, 2>
      `---- 8/27 ---- <3, 3, 3>
```

The first of these converts each bit, 0 or 1, to a sign, -1 or 1. The second computes the sum of five random bits and centers the sum at 0. The third converts each component bit to a sign. The fourth converts each component to 2 if it is even or 3 otherwise. Notice that the weights in the last case are not uniform. These

The last two cases show the kinds' compact forms (see Section 3).

dynamic statistics either as functions or with the `^` operator, interchangeably, `(2 * __ - 1)(bit)` is equal to `bit ^ (2 * __ - 1)`.

Puzzle 17. Use `unfold` to explain the non-uniform weights in the last case. Look at the unfolded kind of `uniform(1,2,3) ** 3` before and after the transformation. Which paths in the former tree correspond to each path in the latter? (Sketching a picture might help.) Now look at the compact trees and see the connection with what you just did.

These dynamic statistics are convenient, but it is sometimes easier to write custom, named statistics. To do this, define a Python function with either `@statistic()` or `@scalar_statistic()` as "decorators" before the definition, with an optional description of the statistic. Remember that our FRPs store lists of **numbers**, so ensure you return numbers (including $\pm\infty$, written as `infinity` and `-infinity`). However, boolean statistics are automatically handled to map False to 0 and True to 1, so it is fine to return a boolean as well. For question 4 above, we might write:

```
playground> @scalar_statistic('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() starts from 0
...>            except:  # There is no roll with value 6, do this
...>                return math.inf
playground> when_first_6(five_d6)
An FRP of dimension 1 and size 6 with value 2.
```

For question 5, we might write:

```
playground> @statistic(name='Number of rolls with most common value')
...>        def most_common_count(rolls):
...>            counts = [0] * len(rolls)
...>            for roll in rolls:
...>                counts[roll] += 1
...>            return max(counts)
playground> most_common_roll(five_d6)
```

```
An FRP of dimension 1 and size 6 with value 2.
```

We can do similar operations on kinds as well to see the *kind* of the transformed FRP. For example:

```
playground> Sum(kind(D) ** 5)
        ,---- 1/7776    ---- 5
        |---- 5/7776    ---- 6
        |---- 15/7776   ---- 7
        |---- 35/7776   ---- 8
        |---- 70/7776   ---- 9
        |---- 126/7776  ---- 10
        |---- 205/7776  ---- 11
        |---- 305/7776  ---- 12
        |---- 420/7776  ---- 13
        |---- 540/7776  ---- 14
        |---- 651/7776  ---- 15
        |---- 735/7776  ---- 16
        |---- 780/7776  ---- 17
<> -|
        |---- 780/7776  ---- 18
        |---- 735/7776  ---- 19
        |---- 651/7776  ---- 20
        |---- 540/7776  ---- 21
        |---- 420/7776  ---- 22
        |---- 305/7776  ---- 23
        |---- 205/7776  ---- 24
        |---- 126/7776  ---- 25
        |---- 70/7776   ---- 26
        |---- 35/7776   ---- 27
        |---- 15/7776   ---- 28
        |---- 5/7776    ---- 29
        `---- 1/7776    ---- 30
```

More on this in subsection 5.2.

Puzzle 18. 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 19. Create a statistic to answer question 6.

With this experience in hand, we can define a statistic more precisely.

Definition 6. A ***statistic*** is a data-processing algorithm, a function that takes as input any of the possible values produced by an FRP and returns a transformed value of a specified type. Here, we will consider only statistics that return lists of numbers, though later we will also consider statistics that return Boolean values (i.e., true and false).

For an FRP of dimension n, a statistic is a function that maps a list of n numbers to a list of n' numbers for some positive integer n'. We call n the ***dimension*** of the statistic[*] and n' the ***co-dimension*** (or codim for short). If the codim is 1, we say it is a *scalar statistic*, and we identify the returned list with the number itself, as we did before.

[*]Sometimes called its *arity*

We will 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").

An FRP and a statistic are ***compatible*** if (i) they have the same dimensions and (ii) every possible value of the FRP is a valid input to the statistic. Analogously, we say that a kind and a statistic are compatible if the same two conditions hold: the dimensions match and the value at each leaf of the kind is a valid input to the statistic.

Put simply: we can plug the output of the FRP into the input of the statistic.

We create a transformed FRP by connecting the All output port of the original FRP to the input port of a trivial FRP using an adapter like that shown in Figure 14. The adapter has circuitry to compute a specific statistic from the value of the original FRP, when it is produced. The original's output port is connected to the bottom of the adapter and the transformed value is emitted from the central output port on top. (The adapter's other two output ports simply copy its input so that we can create multiple transforms of the same original FRP.) The trivial FRP it is connected has its output ports and kind display automatically reconfigured and relabeled accordingly, and its button is disabled. It will display its value after the button on the original FRP is pushed. The connection like this can only be made physically with an adapter for a statistic that is compatible with the original FRP.

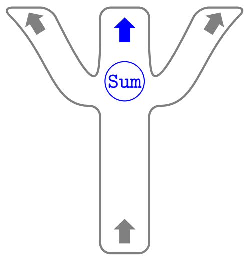


FIGURE 14. An adapter used to transform an FRP by the Sum statistic. The value comes from the original FRPs All output port and is transformed and emitted through the top center port. The original value enters through the bottom port on the adapter and leaves through the top-center port. The other two 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. Notice how that the adapter resembles the letter ψ.

Definition 7. If X is an FRP and ψ is a compatible statistic, then X transformed by ψ – denoted by $\boldsymbol{\psi(X)}$ – is the FRP that produces value $\psi(\langle v_1, \ldots, v_n \rangle)$ when X produces value $\langle v_1, \ldots, v_n \rangle$.

For simple functions like $\psi(x) = x^2$, $\phi(x) = 4x$, or $\zeta(x_1, y_2) = -x_1 x_2$, we can write the transformed FRPs as X^2, $4X$, or $-X_1 X_2$ without naming the statistic.

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. Think of X in this expression as "hole" that we will fill with X's value when it is available.

Note that X^2 is ***not*** the same as $X \star X$; make sure you are clear on why.

Notational Convention. For a function ψ that takes lists of dimension n as input, it is convenient to be flexible with how we write its arguments. If $v = \langle v_1, \ldots, v_n \rangle$, we treat the following as equivalent and interchangeable:

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

using whichever form is clearest and most convenient at any moment.

5.1 Projection and Marginals

Some of the most commonly used statistics are *projections*; these extract one or more components from a list of values into a new list.[*] In the playground, we access these statistics using `Proj[`*indices...*`]`, where the indices start counting from 1. For example:

[*] Remember that we elide the distinction between lists of length 1 and scalars.

```
playground> Proj[3]( (10, 20, 30, 40, 50) )
30
playground> Proj[3,5]( (10, 20, 30, 40, 50) )
(30, 50)
playground> Proj[1,3,5]( (10, 20, 30, 40, 50) )
(10, 30, 50)
```

Between the brackets `Proj` can accept multiple individual indices or a list/tuple/iterable of indices, like `Proj[(1,3,5)]` or `Proj[range(2,5)]`. Note that `range(a,b)` includes `a` but not `b`. When we write these statistics mathematically (as opposed to in `frplib`) we use proj with the indices in a subscript; for instance,

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

In the playground, you can apply a projection statistic directly to an FRP or kind as discussed earlier, but `frplib` also supports direct indexing by index lists or slices (e.g., `1:5:2`). For example, if we make the following definitions,

```
playground> bit = uniform(0,1)
playground> d6 = uniform(1,2,3,4,5,6)
playground> the_bits = [1, 3, 5]
playground> also_the_bits = Proj[1, 3, 5]
playground> first_d = Proj[2]
```

the following are equivalent:

```
playground> (bit * d6 * bit * d6 * bit) ^ Proj[1, 3, 5]
playground> (bit * d6 * bit * d6 * bit) ^ also_the_bits
playground> (bit * d6 * bit * d6 * bit)[1, 3, 5]
playground> (bit * d6 * bit * d6 * bit)[the_bits]
playground> (bit * d6 * bit * d6 * bit)[also_the_bits]
playground> (bit * d6 * bit * d6 * bit)[1:5:2]
```

where `1:5:2` is a "slice" saying "go from 1 to 5 skipping by 2" and where you can use lists of indices or Projection statistics as well as explicit integers. Similarly, the following are equivalent

```
playground> Proj[2](bit * d6 * bit * d6 * bit)
playground> first_d(bit * d6 * bit * d6 * bit)
playground> (bit * d6 * bit * d6 * bit)[2]
playground> (bit * d6 * bit * d6 * bit)[first_d]
```

Try these out and see the results.

Definition 8. If X is an FRP and $X' = \mathsf{proj}_{i_1,i_2,\ldots,i_m}(X)$ 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 $i_1, i_2, \ldots, i_m$.

If k is a kind and $k' = \mathsf{proj}_{i_1,i_2,\ldots,i_m}(k)$ 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 $i_1, i_2, \ldots, i_m$.

The process of transforming an FRP or kind this way is called ***marginalization***.

When dealing with high-dimensional FRPs, we often want to refer to the value at some index in the list of numbers produced, so it helps to have some language for that. 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 $i \in [1\,..\,n]$, 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 $\mathsf{proj}_i(X)$. If we define $X_i = \mathsf{proj}_i(X)$ for $i \in [1\,..\,n]$, we call $\langle X_1, X_2, \ldots, X_n\rangle$ the the *components* of X.

Using the previous (somewhat silly) example,

```
playground> random_stuff = frp(bit * d6 * bit * d6 * bit)
playground> random_stuff
An FRP of dimension 5 and size 288 with value <0, 4, 1, 2, 0>.
```

```
playground> random_stuff[1]
An FRP of dimension 1 and size 2 with value 0.
playground> random_stuff[2]
An FRP of dimension 1 and size 6 with value 4.
playground> random_stuff[3]
An FRP of dimension 1 and size 2 with value 1.
playground> random_stuff[4]
An FRP of dimension 1 and size 6 with value 2.
playground> random_stuff[5]
An FRP of dimension 1 and size 2 with value 0.
```

Puzzle 20. If S, T, U are scalar FRPs and $W = S \star T \star U$, what are the components of W?

Puzzle 21. If X has components $\langle X_1, X_2, \ldots, X_n \rangle$, is it always true that we can write $X = X_1 \star X_2 \star \cdots \star X_n$? If so why? If not, can you find an example where this relationship does not hold?

Definition 9. If an FRP X has dimension n, then we can decompoese it into ***components***, *scalar* FRPs $X_1, \ldots, X_n$ with $X_i = \mathsf{proj}_i(X)$ for each $i \in [1\,..\,n]$. We call $X_1, \ldots, X_n$ the components FRPs of X, or just the components of X for short.

5.2 Transformed Kinds

Throughout our exploration of FRPs so far, we've seen a close parallel between FRPs and their kinds. A mixture of FRPs, for example, has a kind that is the mixture of the original kinds. The same relationship holds with transformation by statistics.

If X is an FRP with kind k and ψ is a compatible statistic, then the transformed FRP $\psi(X)$ has a kind that we denote by $\psi(k)$.

The transformation of FRPs makes concrete sense: we get a value from the device and pass that vaule as input to the statistic. For kinds, the transformation seems a bit more abstract, so let's start with the picture in Figure 15 to help understand it. The top panel of the figure shows a kind of dimension 3 in unfolded form that we want to transform by the `Sum` statistic. The blue bar represents the action of the statistics adapter: each value is mapped to the sum of its components. These will be the values of the new, transformed FRP, but remember that all the values of an FRP must be *distinct*. So, we need to combine all the paths in this tree with equal leaf.

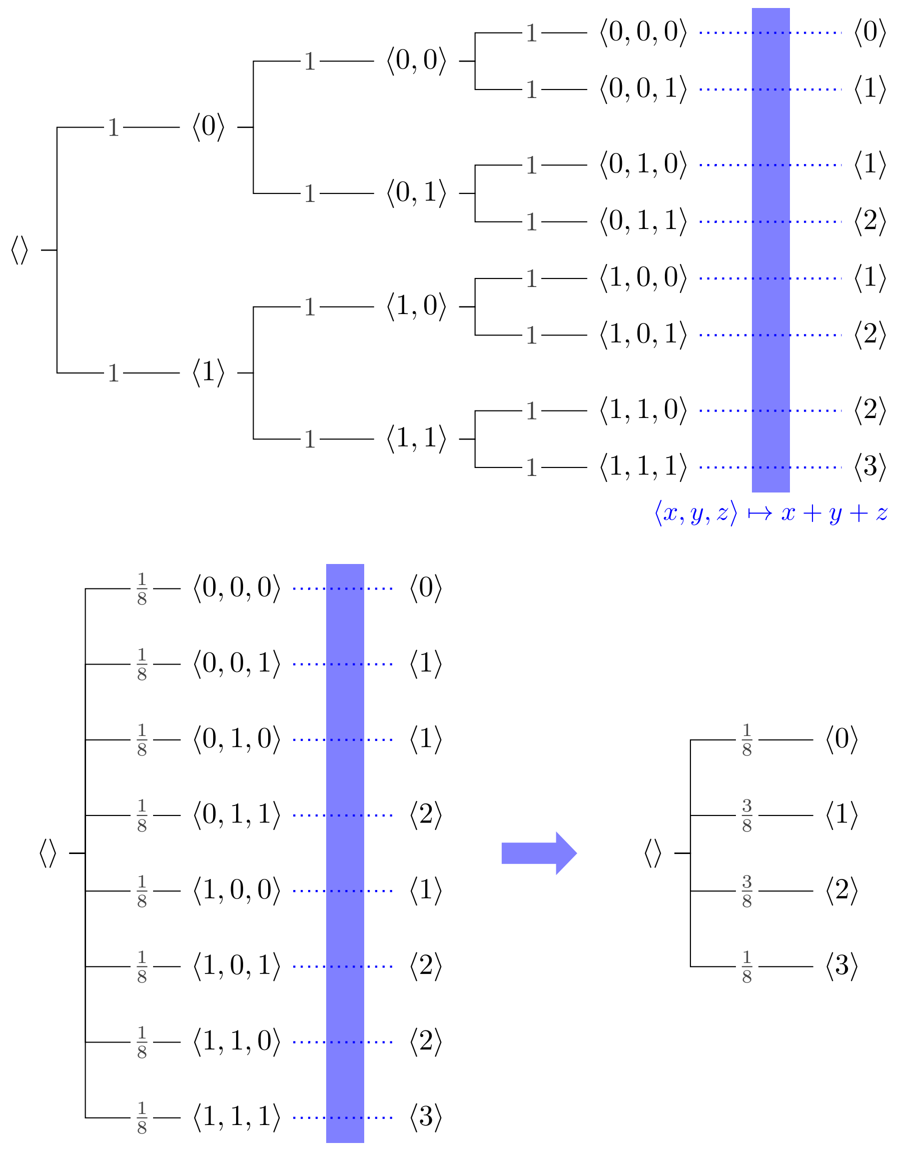
To do this, we first convert the tree to canonical form, as described in Section 3. This gives us the weights on each path in a way that is comparable across the tree. This tree is shown on the bottom left of the figure. The final step is to combine the paths from the canonical tree with the same leaf, adding the canonical weights. The result is the transformed tree, at bottom right.

Figure 16 is pretty much the same story. The only difference here is that there are no "duplicates," so the mapping is direct in the bottom of the panel.

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.
- Use the playgroud 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.
- Describe what it means for a statistic to be compatible with an FRP or kind.



$\langle\rangle$
1
$\langle 0\rangle$
$\langle 1\rangle$
$\langle 0,0\rangle$
$\langle 0,1\rangle$
$\langle 1,0\rangle$
$\langle 1,1\rangle$
$\langle 0,0,0\rangle$ $\langle 0\rangle$
$\langle 0,0,1\rangle$ $\langle 1\rangle$
$\langle 0,1,0\rangle$ $\langle 1\rangle$
$\langle 0,1,1\rangle$ $\langle 2\rangle$
$\langle 1,0,0\rangle$ $\langle 1\rangle$
$\langle 1,0,1\rangle$ $\langle 2\rangle$
$\langle 1,1,0\rangle$ $\langle 2\rangle$
$\langle 1,1,1\rangle$ $\langle 3\rangle$
$\langle x,y,z\rangle \mapsto x+y+z$
$\frac{1}{8}$
$\frac{3}{8}$

FIGURE 15. (Top) A kind (`bit ** 3` in the playground) with its output connected to the `Sum` statistic. The blue bar identifies where the statistics adapter takes effect, and the $\langle x, y, z\rangle \mapsto x + y + z$ label shows how the statistics acts on the values of the original kind. (The label could also be just the word "Sum" in this case.) The values to the right of the blue bar are the output values for the transformed FRP. (Bottom Left) The original kind in canonical form. To obtain the transformed kind, we combine all leaves with the same value, adding their canonical weights. (Bottom Right) The transformed kind from the other panels in canonical form.

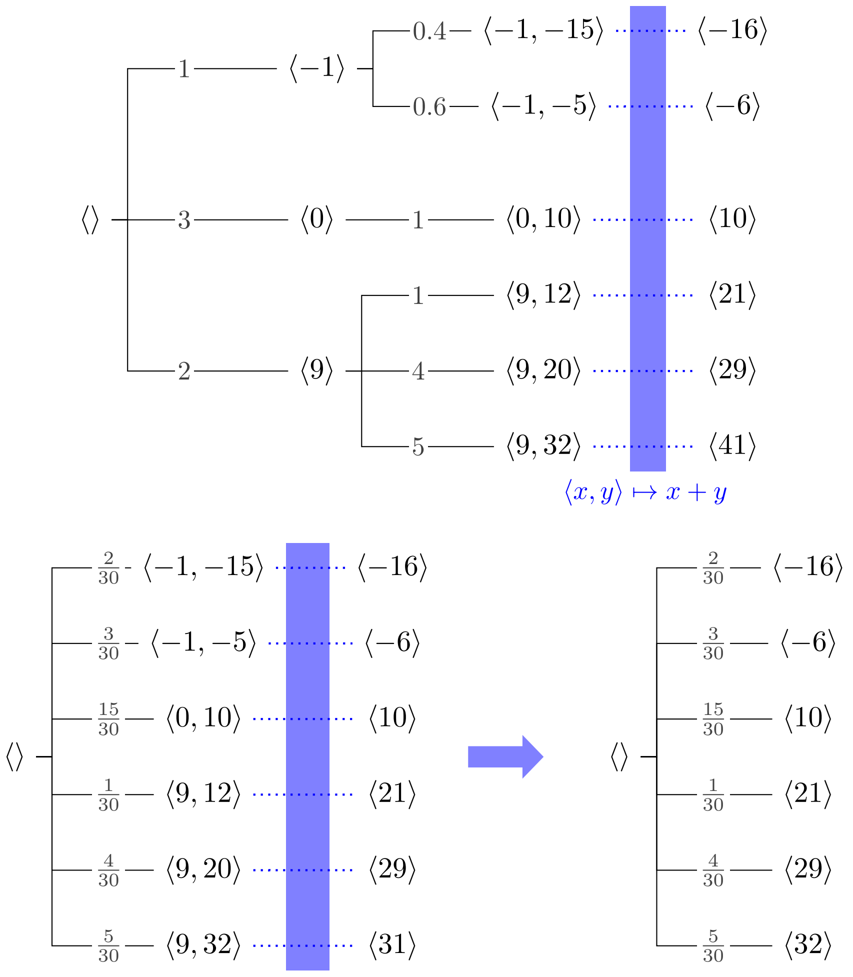


FIGURE 16. (Top) A kind with its output connected to the `Sum` statistic. The blue bar identifies where the statistics adapter takes effect, and the $\langle x, y\rangle \mapsto x + y$ label shows how the statistics acts on the values of the original kind. (The label could also be just the word "Sum" in this case.) The values to the right of the blue bar are the output values for the transformed FRP. (Bottom Left) The original kind in canonical form. To obtain the transformed kind, we combine all leaves with the same value, adding their canonical weights. (Bottom Right) The transformed kind from the other panels in canonical form.

6 Making New FRPs: Constraining with Conditionals

Key Take Aways

We can combine FRPs by connecting output ports to input ports in several ways to build new FRPs. The resulting FRPs capture important concepts: generating random outcomes contingent on earlier outcomes (mixtures), transforming the output of an FRP by some algorithm (statistics), and making predictions based on partial information (conditionals). The operations on FRPs lead to directly analogous operations on kinds, which will be fundamental in building and analyzing models for real systems. By combining mixtures, statistics, and conditionals we can model – and answer – most questions that arise in probability theory.

A ***conditional*** applies when we have partial information about the value of an FRP. In this case, we can express the uncertain information as a new, reduced FRP that returns a value *consistent with our partial information.* Conditionals capture the phenomenon of making observations as a random process unfolds and updating our predictions.

You run into Alice and Bob at the FRP market, and Alice is agitated. It seems that she had planned to place an order for a sizeable batch of b FRPs $A_{[1]}, A_{[2]}, \ldots, A_{[b]}$ with the kind shown in Figure 17, where the deal stipulates that Alice will receive payoff from $P_{[i]} = \mathsf{proj}_3(A_{[i]})$ for $i \in [1\,..\,b]$. Anticipating her likely order – she is a regular customer – the market staff inadvertently pressed the buttons on $A_{[1]}, \ldots, A_{[b]}$ early, before she had committed to the deal. And Bob, who was wandering about the facility happened to catch a glimpse of the displays. He told Alice what he saw, and Alice proceeded with the order, happily. When the market administrator heard what happened, she tried to rescind the order, which led to lawyers getting involved. Let's see why Alice was happy with the order and frustrated with the administrator's response.

Bob has a good recall, or a fast camera!

Some specifics: from his vantage point, Bob could see only the *second* number in the list displayed by each FRP $A_{[i]}$. He quickly tabulated up what he observed and showed it to Alice. Out of 10,000 FRPs, he saw 4850 0's and 5150 1's. Alice perked up at the news, convinced that the negotiated price for her order was now a bargain.

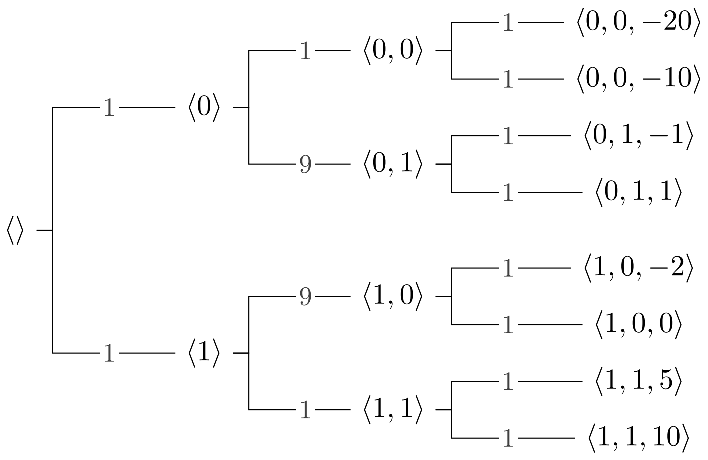


FIGURE 17. The kind for the FRPs in Alice's order, *before* she learned Bob's information.

Why was Alice so happy? What does the partial information that Bob oberved mean to her? The key observation here is that once Alice obtains knowledge that one component is fixed, the FRP that determines her payoff has *effectively* been changed. To understand this, it will be useful to focus on a single FRP of the same kind (Figure 17) as $A_{[1]}, \ldots, A_{[b]}$.

Suppose you have such an FRP, call it A, and you learn or observe that the second component has value 1. You do not observe the first component, but you have nonetheless obtained information about it implicitly. To see why, look at all the paths in the 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 18.

The trick to doing so cleanly is to first reduce to canonical form.
Figure 19 shows the original kind in canonical form, highlighting the paths that are consistent with the available information. To account for the information, we simply drop the inconsistent branches of the tree and combine branches with the same value, adding their weights. A quick rescaling (useful but not required) returns us to canonical form, giving the kind in the top panel of Figure 20. This is the kind of the FRP that Alice effectively gets ***conditional*** on the information that the second component is 1. A similar argument yields the bottom panel of Figure 20, which

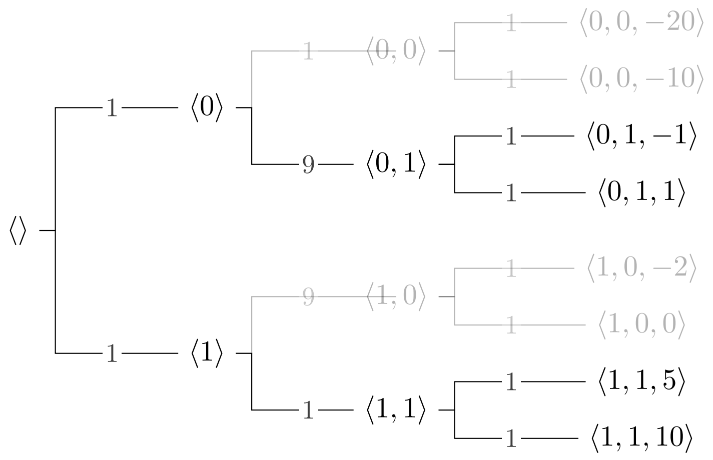


FIGURE 18. The kind from the previous Figure, highlighting the paths in the tree that are consistent (black) and inconsistent (gray) with the observation that the second component of the value equals 1.

shows the effective FRP conditional on observing that the second component is 0. (You can obtain it by following the same procedure with different subtrees; just use the gray parts of the previous Figure.)

Notice that all the values in Figure 20'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, but some values are eliminated from consideration in this case.

We denote the effective FRP corresponding to the top panel by

$$A \mid \mathsf{proj}_2(A) = 1 \tag{6.1}$$

The | bar here reads as "conditional on," or more typically, "given." The statement after the | is the condition that we have observed to be true. If we let $\langle A_1, A_2, A_3\rangle$ be the component FRPs of A, we could also write this as

Recall that this means $A_i = \mathsf{proj}_i(A)$.

$$A \mid A_2 = 1. \tag{6.2}$$

If we transform this conditional FRP by a statistic ψ, it is the same as transforming the original FRP and then applying the conditional, so

$$\psi(A \mid A_2 = 1) \equiv \psi(A) \mid A_2 = 1 \tag{6.3}$$

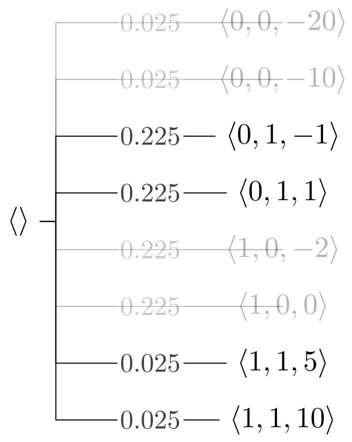


FIGURE 19. The canonical form of the kind in the previous Figure, highlighting the values that are consistent (black) and inconsistent (gray) with the observation that the second component of the value equals 1.

and we write it in the latter form in practice.

Since Alice put a price on the statistic $\psi = \mathsf{proj}_3$ in purchasing these FRPs, we can compare the kinds of $A_3 \mid A_2 = 1$ by transforming the top kind in Figure 20. Figure 21 compares the kinds of $A_3 \mid A_2 = 1$ (left) and A_3 (right). The reason Alice was happy to proceed with the order when she learned Bob's information (and unhappy when the administrator moved to rescind the order) is that the kind at left is *worth more* than the kind at the right.

In the next section, we will learn how to attach a fair price to an FRP and in turn learn to predict a typical value, or "expectation," of its payoff. But for now, we will can use the `frp playground` to compute it.

Run the `frp playground` application and follow along. Parts of the line after `#` are comments for your benefit, but you *need not* type them in. To keep things simple, we'll load the kind $\mathsf{kind}(A)$ from the library. I will not show all the output here, but you can check your results by the information in the text and figures above.

```
playground> from frplib.examples.alice_bob import kindA
playground> kindA                       # Shows the canonical form
playground> A = frp(kindA)              # An FRP with that kind
playground> A | (Proj[2] == 1)          # The FRP given Bob's info
playground> kind(A | (Proj[2] == 1)) # Look familiar?
```

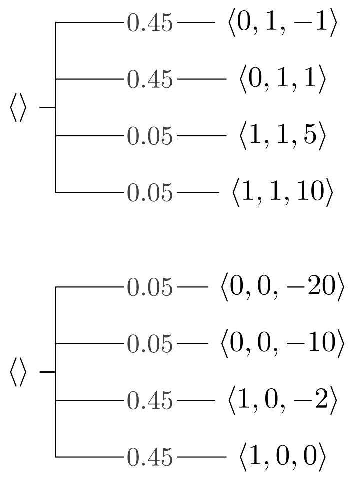


FIGURE 20. The *effective* kind of Alice's FRPs conditional on the information that the second component of the value equals 1 (top) and 0 (bottom).

```
playground> kindA | (Proj[2] == 1)   # It works on kinds too!
```

It might not be surprising that operations on FRPs are mirrored by analogous operations on kinds, and vice versa. In fact, if you go through the argument above, we actually defined $\mathsf{kind}(A \mid A_2 = 1)$ by $\mathsf{kind}(A) \mid A_2 = 1$.

To find a good price – and hence a good prediction – of the FRPs value, we will use a new operator, E:

```
playground> E(A[3])                # Risk-neutral price for A[3]
-0.825
playground> E(A[3] | (Proj[2] == 1)) #...and for A[3] given A[2] == 1
0.75
playground> E(A[3] | (Proj[2] == 0)) #...and for A[3] given A[2] == 0
-2.4
playground> E(A[2])                # Risk-neutral price for A[2]
0.5
```

This means that Alice had originally contracted for a unit price $-0.825 as that was her predicted/typical/"expected" payoff from each FRP, but with Bob's information her predicted/typical/"expected" payoff from each FRP is now $0.75. She stands to make a lot of money.

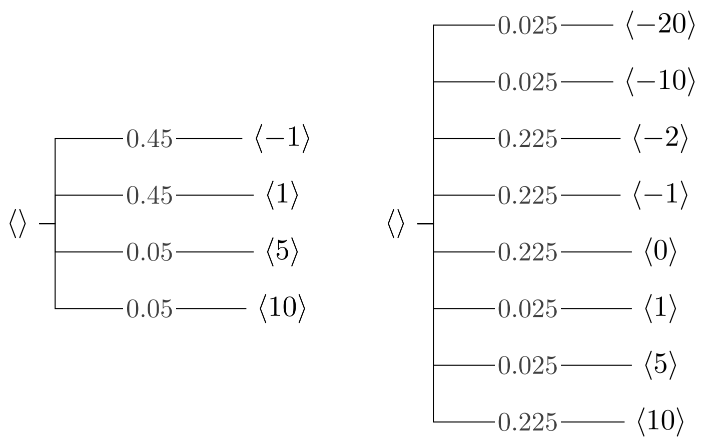


FIGURE 21. The *effective* kind of Alice's FRPs conditional on the information that the second component of the value equals 1 (top) and 0 (bottom).

Recall that `A[3]` is just $A_3 = \mathsf{proj}_3(A)$ and similarly for `A[2]`. Mathematically, we write these statements as $\mathbb{E}(A_3)$, $\mathbb{E}(A_3 \mid A_2 = 1)$, $\mathbb{E}(A_3 \mid A_2 = 0)$, and $\mathbb{E}(A_2)$. It is not a coincidence that $-0.825 = 0.5 \cdot 0.75 + 0.5 \cdot -2.4$ and $\mathbb{E}(A_2) = 0.5$. We will see what this means and where it comes from in the next section. For now, we turn to defining conditionals in general.

An FRP given a condition is an FRP derived from another by the assertion that some information has been observed with certainty. We write a conditional with the form using $\mathit{FRP} \mid \mathit{Condition}$, where the $\mid$ is read as "conditional on" or "given" and *Condition* specifies the information that is asserted as known.

Conditions are just Boolean expressions that associate "true" or "false" to each possible value of an FRP (and leaf node in a kind). That means, conditions are just *statistics* that return Boolean values.

Definition 10. A *condition* is a statistic that returns a Boolean value.

A Boolean value is either $\top$, read "true" or "top", or $\bot$, read "false" or "bottom".

We also abuse notation a little bit by letting $\top$ and $\bot$ denote *constant* conditions. In this case, $\top$ denotes the constant condition that returns true for *any* value and $\bot$ denotes the constant condition that returns false for any value.

In our *computations*, we often identify $\top$ with the number 1 and $\bot$ with the number 0, but we distinguish them notationally because Boolean and number are conceptually distinct types.

Recall the definition of *compatibility* between a statistic and an FRP or kind given in Section 5. The dimensions must match and the values of the FRP/kind must all be valid inputs to the statistic. The same definition applies here, so we can speak of *conditions* that are compatible with FRPs and kinds.

The good news is that most of the conditions we work can be written in a simple and intuitive form. We saw several conditions in the Alice and Bob example earlier. For instance, in playground we wrote `Proj[2] == 1` for the statistic that takes a value and returns $\top$ if the second component of the value equals 1. (Notice that we use `==` for the equality relation in the playground.) Mathematically, this is just a function from values to Booleans, but we usually write it in a more intuitive way: either,

$$\mathsf{proj}_2(\cdot) = 1$$

or when working with a particular FRP like A,

$$\mathsf{proj}_2(A) = 1.$$

These expressions describe the condition with $\cdot$ and the FRP name acting as a "hole" into which the FRP's value will be inserted when available to compute the value of the condition. When dealing with a particular FRP, we usually use the second form because the name helps remember what we are dealing with. In the playground, we can use `__` as a similar "hole," e.g., `2 * __ + 1`, though as the earlier examples show, it is not always necessary to do so.

We can combine conditions 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, $\mathsf{proj}_2(A) = 1 \wedge \mathsf{proj}_1(A) = 1$ and $\mathsf{proj}_2(A) = 1 \text{ and } \mathsf{proj}_1(A) = 1$ mean the same thing. In the playground, things are a little more awkward because of restrictions of the host language (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 tests. 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.

Definition 11. If k is a kind, then the ***kind given a condition*** ζ, denoted by $k \mid \zeta$, is the kind obtained from k by eliminating all paths inconsistent with ζ, that is all paths that lead to values v with $\zeta(v) = \bot$. Both k and $k \mid \zeta$ have the same dimension.

If X is an FRP, then the ***FRP given a condition*** ζ, denoted by $X \mid \zeta$, is the FRP obtained from X that forbids any value for which *zeta* returns $\bot$. Both X and $X \mid \zeta$ have the same dimension.

We have the key relationship:

$$\mathsf{kind}(X \mid \zeta) = \mathsf{kind}(X) \mid \zeta. \tag{6.4}$$

Notice also that

$$k \mid \top = k \qquad\qquad X \mid \top = X \tag{6.5}$$

$$k \mid \bot = \langle\rangle \qquad\qquad X \mid \bot = \mathsf{empty}, \tag{6.6}$$

because observing $\top$ does not eliminate any branches (since they all are consistent with true) and observing $\bot$ eliminates all of them (since none of them are consistent with false). So a kind or FRP without a conditional is equivalent to a conditional on a trivially true condition.[*]

[*] Say that ten times fast.

If we observe some partial information about the value of an FRP, then we just eliminate from consideration any possible values that are inconsistent with the observed information.

Let's look a couple more concrete examples in the playground. Follow along and look at the output even when I do not reproduce it here. First, consider FRPs that generate random bits.

As usual, when 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.

```
playground> bit = uniform(0,1)              # 0 or 1 with equal weight
playground> biased_bit = either(0,1,9)     # 0 or 1 with 9:1 weight ratio
playground> three_random_bits = bit ** 3
playground> three_biased_bits = biased_bit ** 3
```

Puzzle 22. What do you expect to see when you enter

```
three_random_bits | (__ == (1, 0, 1))
```

in the playground? Explain why this makes sense.

Let's first try several conditionals derived from `three_random_bits`.

```
playground> three_random_bits
    ,---- 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>
playground> three_random_bits | (Sum == 2)
    ,---- 1/3 ---- <0, 1, 1>
<> -+---- 1/3 ---- <1, 0, 1>
    `---- 1/3 ---- <1, 1, 0>
playground> three_random_bits | (Proj[1] == Proj[2])
    ,---- 1/4 ---- <0, 0, 0>
    |---- 1/4 ---- <0, 0, 1>
<> -|
    |---- 1/4 ---- <1, 1, 0>
    `---- 1/4 ---- <1, 1, 1>
playground> three_random_bits | (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>
```

In each of these conditionals, you can directly trace the method we used to derive the conditional: simply eliminate any leaves that are inconsistent with the condition and then (optionally but often helpfully) return to canonical form – so the weights sum to 1. Notice the specified condition holds for all the values in the resulting kind.

The next two examples are similar but apply an additional transformation to the conditional.

```
playground> ( three_random_bits | And(Proj[1] == 1, Proj[2] == 0) )[3]
     ,---- 1/2 ---- 0
<> -|
     `---- 1/2 ---- 1
playground> ( three_random_bits | (Proj[1] != Proj[2]) ) ^ (Proj[1] + Proj[3])
     ,---- 1/4 ---- 0
<> -+---- 1/2 ---- 1
     `---- 1/4 ---- 2
```

The first result here looks like the kind of a random bit. This is not a coincidence because `three_random_bits` is an *independent mixture* – the third bit's value is generated the same way regardless of the other bits – and the condition does not reference the third bit. Compare this with the output of `three_random_bits[3]` without the conditional.

Puzzle 23. Suppose `psi` is a statistic in the playground that only depends on the first two bits. What do you expect to see when you compare the following two kinds?

```
( three_biased_bits | psi )[3]
three_biased_bits[3]
```

Answer the same question substituting `three_random_bits` in both expressions.

Explain this.

Next, we revisit Example 4.2 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.

```
playground> has_disease = either(0, 1, 999)     # No disease has higher weight
playground> test_by_status = conditional_kind({
...>            (0,): either(0, 1, 99),  # No disease, negative has high weight
...>            (1,): either(0, 1, 1/19) # Yes disease, positive higher weight
...>        })
playground> dStatus_and_tResult = has_disease >> test_by_status
playground> dStatus_and_tResult

    ,---- 98901/100000 ---- <0, 0>
    |---- 999/100000   ---- <0, 1>
<> -|
    |---- 1/20000      ---- <1, 0>
    `---- 19/20000     ---- <1, 1>

playground> Disease_Status = Proj[1]     # Naming this statistic
playground> 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 do we know about the `Disease_Status` component?

Puzzle 24. 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 on the observed information of test result and a projection onto disease status.

```
playground> (dStatus_and_tResult | (Test_Result == 1))[Disease_Status]
```

```
      ,---- 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.

This is an example of what we will later call *Bayes's Rule.* The surprisingly small weight on having the disease even with a positive test result derives from the low baseline prevalance in the population. Work out carefully how this result was derived. The amazing thing is how simple it is; we just exclude branches and renormalize. The marginal on `Disease_Status` of course 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 marginalization to see this.)

Recall that when we discussed this example in Section 4 on mixtures, we called `test_by_status` a *conditional kind* because it specifies a kind contingent on some other specified value. The uses of "conditional" here and there are directly connected. For instance, when you evaluate

```
  (dStatus_and_tResult | (Disease_Status == 0))[Test_Result]
  (dStatus_and_tResult | (Disease_Status == 1))[Test_Result]
```

you will see that these are just `test_by_status[0]` and `test_by_status[1]`, respectively. Notice also that

```
  dStatus_and_tResult[Disease_Status] == has_disease
```

since `has_disease` is just the top level of the unfolded `dStatus_and_tResult`.

In general, if k is a kind of dimension d_k and m is a conditional kind of dimension d_m, then $k \triangleright \mathsf{m}$ has dimension $d_k + d_\mathsf{m}$. From any *value* of $k \triangleright \mathsf{m}$, we can extract the k value with `Proj[1:d_k]` and the corresponding `m` value with `Proj[(d_k+1):(d_k+d_m)]`. Define

Recall that `Proj[a:b]` is a statistic that extracts components $a, a+1, \ldots, b$. In math we write this as $\mathsf{proj}_{a:b}$.

```
  playground> ks_values = Proj[1:d_k]
  playground> ms_values = Proj[(d_k+1):(d_k+d_m)]
```

Then for every value v of k:

```
    k == (k >> m)[ks_values]
    m(v) == (k >> m | (Proj[ks_values] == v))[ms_values]
```

In other words, for the conditional kind m, $\mathsf{m}(v)$ is the *kind given the condition* that k's value equals v. So m just packages all the conditionals given each value of k.

We can state this precisely in mathematical terms very much the same way.

Suppose k is a kind of dimension d_k and m is a compatible conditional kind with dimension d_m, then $k \triangleright \mathsf{m}$ has dimension $d_k + d_{\mathsf{m}}$. . Then,

$$k = \mathsf{proj}_{1:d_k}(k \triangleright \mathsf{m}) \tag{6.7}$$

and for every value v of k,

$$\mathsf{m}(v) = \mathsf{proj}_{(d_k+1):(d_k+d_{\mathsf{m}})}(k \triangleright \mathsf{m} \mid \mathsf{proj}_{1:d_k}(\cdot) = v). \tag{6.8}$$

Checkpoints

After reading this section you should be able to:

- Define a condition and construct several examples, mathematically and in the playground.
- Define a *kind given a condition* and an *FRP given a condition.*
- Describe how to find $k \mid \zeta$ if you are handed a kind k, in canonical form, and a condition ζ.
- Identify the difference, if any, between $k \mid \top$ and k for a kind k.

7 Interlude: Computations in Practice

Key Take Aways

This section highlights several techniques for doing computations in the playground and gives some interesting example calculations. The context is a friendly conversation between Alice and Bob.

Alice and Bob are clients of the FRP warehouse whom you met during the orientation for new users. This conversation took place during a workshop then.

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

$$\begin{aligned}
\mathsf{Sum}(\langle 10,20,30\rangle :: \langle 40,50\rangle) &= \mathsf{Sum}(\langle 10,20,30\rangle) + \mathsf{Sum}(\langle 40,50\rangle)\\
&= \mathsf{Sum}\left(\left\langle \mathsf{Sum}(\langle 10,20,30\rangle), \mathsf{Sum}(\langle 40,50\rangle)\right\rangle\right),\\
\mathsf{Sum}(\langle 10,20\rangle :: \langle\rangle) &= \mathsf{Sum}(\langle 10,20\rangle) + \mathsf{Sum}(\langle\rangle)\\
&= \mathsf{Sum}\left(\left\langle \mathsf{Sum}(\langle 10,20\rangle), \mathsf{Sum}(\langle\rangle)\right\rangle\right),
\end{aligned}$$

because the sum of an empty list is 0. Make sense?

Bob: That's a mouthful, but yes, I see. $\mathsf{Sum}(a :: b) = \mathsf{Sum}\left(\left\langle \mathsf{Sum}(a), \mathsf{Sum}(b)\right\rangle\right)$. 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

```
playground> sum_of_4_rolls = Sum(d6 ** 4)
playground> sum_of_100_rolls = sum_of_4_rolls    # initialize
playground> for iter 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
|---- 1/324    ---- 5
|---- 5/648    ---- 6
|---- 5/324    ---- 7
|---- 35/1296  ---- 8
|---- 7/162    ---- 9
|---- 5/81     ---- 10
|---- 13/162   ---- 11
|---- 125/1296 ---- 12
```

```
    |---- 35/324   ---- 13
<> -+---- 73/648   ---- 14
    |---- 35/324   ---- 15
    |---- 125/1296 ---- 16
    |---- 13/162   ---- 17
    |---- 5/81     ---- 18
    |---- 7/162    ---- 19
    |---- 35/1296  ---- 20
    |---- 5/324    ---- 21
    |---- 5/648    ---- 22
    |---- 1/324    ---- 23
    `---- 1/1296   ---- 24
```

If we mix it with itself, `sum_of_4_rolls * sum_of_4_rolls`, it 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,

$$\begin{aligned} \min(\langle 10,20,30\rangle :: \langle 40,50\rangle) &= \mathsf{Min}\left(\left\langle\, \mathsf{Min}(\langle 10,20,30\rangle), \mathsf{Min}(\langle 40,50\rangle)\,\right\rangle\right) \\ \min(\langle 10,20\rangle :: \langle\rangle) &= \mathsf{Min}\left(\left\langle\, \mathsf{Min}(\langle 10,20\rangle), \mathsf{Min}(\langle\rangle)\,\right\rangle\right), \end{aligned}$$

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 $\mathsf{Min}(a :: b) = \mathsf{Min}\left(\left\langle\, \mathsf{Min}(a), \mathsf{Min}(b) \,\right\rangle\right)$.

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, $\mathsf{Fork}(s_1, s_2)(x) = s_1(x) :: s_2(x)$ and $\mathsf{Fork}(s_1, s_2, s_3)(x) = s_1(x) :: s_2(x) :: s_3(x)$.

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 with the `lazy` operator, 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 though 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

```
playground> 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 *i*nclusive range of *i*ntegers from its first to second arguments, with an option to exclude values in a set. This looks like

```
playground> 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, so we just return a kind rather than a conditional kind. In (2), we define the conditional kind that we are going to return as a function; this takes in the values of the earlier kind, which means a list of all previous cards. In (3), we get 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 return the conditional kind, using `conditional_kind` to give us nicer output if we look at it.

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

```
playground> card(1) >> card(2)
playground> card(1) >> card(2) >> card(3)
playground> 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

```
playground> shuffle = card(1)
playground> 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 fine, but I'm looking for an *idea* for understanding 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 heppens 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 conditional for the sixth card given a specific five cards in my hand.

```
playground> my_hand = card(1) >> card(2) >> card(3) >> card(4) >> card(5)
playground> next_card = (my_hand >> card(6) | (Proj[1:5] == (16,17,18,19,4)))[6]
playground> next_card ^ Or(__ == 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.

```
playground> d6_frp = frp(d6)
playground> 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

```
  playground> deck = draw(1)
  playground> for n in irange(2, 52):
  ...>            deck = deck >> draw(n)
  playground> 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([1,3,2])` swaps the second and third components in a value list.

```
  playground> psi = Permute([1,3,2])
  playground> 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 "..."

```
  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.

```
cards(1) ^ Permute(...) == cards(1)
```

BOB: In other words, doing a permutation of 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

```
playground> my_hand = (c_1, c_2, c_3, c_4, c_5)
playground> constant(my_hand) >> cards(6) >> cards(7) >> cards(8)
```

ALICE: Nice. You used the fact that

```
playground> first_five = cards(1) >> cards(2) >> cards(3) >> cards(4) >> cards(5)
playground> 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 25. 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 an FRP that outputs 1 if the pattern occurs and 0 otherwise.

For the next puzzles, we refer to the following example.

Example 7.1. 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 22.

If you are familiar with *regular expressions*, this language is described by `a+(,a+)*`.

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 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.

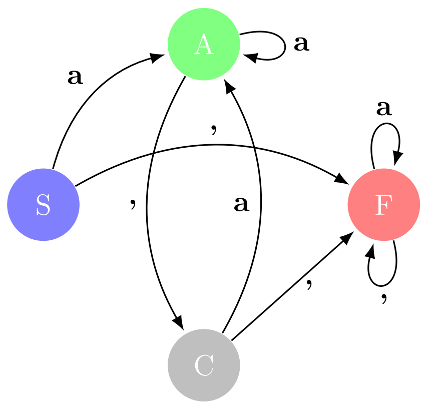


FIGURE 22. The language described in Example 7.1.

Puzzle 26. Referring to the situation in Example 7.1, 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 27. We are interested in whether the string produced by `char ** 80` in the previous puzzle belongs to the language described by Example 7.1.

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.

Checkpoints

After reading this section you should be able to:

- Identify situations where FRP sizes grow quickly.
- Use conditional kinds/FRPs to describe steps in a process.
- For some large kinds, find an efficient way to answer targeted questions.

8 Risk-Neutral Prices and 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.

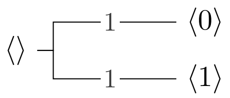
The set of arbitrage prices for an FRP contains every number from $-\infty$ up to ***but not including*** a value r, which may be a real number or ∞. This value r is the ***risk-neutral price*** for the FRP. It is the smallest value that is bigger than all arbitrage prices for that FRP.

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.

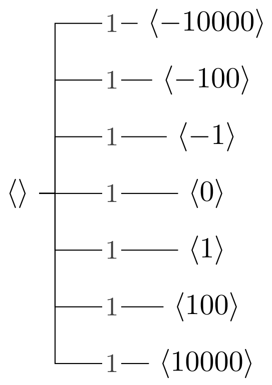
The term risk-neutral here means that the price is not sensitive to the risk of loss that you face or the degree of uncertainty in the FRPs value. When you can purchase as many FRPs as you like with interest-free funding, you are not sensitive to risk as we would be in real life. As such, the risk-neutral price reflects our best prediction of a "typical value" produced by the FRP. More on this point later.

We can use our free trial at the FRP market to estimate the risk-neutral prices for any FRP. We will see an easy way to calculate it in ATTN

The more we know about a phenomenon, the better we are able to predict its outcome. 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 certainty that 100 is the value it will display when we push its button. Close to that is what we will 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.

Approaching the other extreme, an outcome may be uncertain with nothing to distinguish the possibilities. For instance, the FRP with kind 

can produce values 0 or 1, but as we have seen earlier, there is no reason to expect one more than the other. I say “approaching” here because we can add increase our uncertainty by having more and more spread out possible values. For instance, an FRP with kind



seems even harder to predict. And we can increase that uncertainty without bound with ever more complicated kinds.

This raises three questions. First, what prediction should we make 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 in this course.

Our goal in 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.

And the companies have armies of analysts, called actuaries, whose job is to make those predictions based on the available data.

The last example has a resemblance to what we face with FRPs. An insurance company makes their money *in the aggregate*: not on an individual homeowner's premiums but on a large collection of similar premiums. Particular homeowners may require payouts on their policies, but with enough policies and good predictions, the company can price the premiums high enough to make a profit.

This assumes that the different policies are close to what we will call *independent*; if all of the homes are hit by the same hurricane, the company will lose.

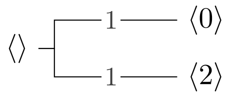
Similarly, with one particular FRP, we can get any of its possible payoffs. But as we saw in the demos of the last section, with a large enough collection of FRPs of the same kind, we will see all of its possible payoffs in some proportion, and we can better control our gains and losses.

We will represent our prediction of an FRPs value though the ***risk-neutral price*** for each FRP of that kind, to be defined below. Here is the setup. We have through the FRP market an unlimited collection of FRPs of any kind. 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. We can borrow as much money as we like interest-free to purchase FRPs, but when their payoffs are revealed, we must immediately pay back that loan.

Remember that negative payoffs means that we have to pay *out*.

We will define the ***risk-neutral price*** c^*_k below, for any FRP of kind k. We will see that you would be unwilling to pay more than c^*_k for each FRP, and in our setting (where you have access to unlimited interest-free funds), the market will not be willing to sell you an FRP for less. The term "risk-neutral" has a meaning that will be clarified below; loosely, it is the price that someone is willing to pay who is indifferent to the magnitude of uncertainty in an FRP's payoff.

Let us consider the simplest, non-trivial FRP: the constant, scalar FRP: the constant $\langle\rangle$ ——— $\langle v\rangle$ 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$. For all practical purposes, this constant FRP is equivalent to \$$v$, which is its risk-neutral price.

Consider next the simple FRP 

Use the `frp market` application to purchase collections of these at different prices. With the `buy` command, 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 `buy` command's and some sample output.

Remember, this is still your free trial, so no money changes hands yet.

```
market> 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

market> 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
```

```
market> 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

market> 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:

```
market> 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-05
```

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.

But now it is time to define our terms. We will define the risk-neutral price relative to prices for an FRP that let us make money with essential certainty.

Definition 12. If X is a scalar FRP, an ***arbitrage price*** for X is a real value c such that if you pay \$$c$ per FRP, you can purchase a collection of FRPs of kind equivalent to $\mathsf{kind}(X)$ 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 example with prices less than 1. So, *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. We can start with some immediate intuition: if c is an arbitrage price for X and $c' < c$, then c' is also an arbitrage price. 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, 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 1 is an arbitrage price for a particular FRP. Property A tells us that we can find an arbitrage price that is slightly bigger than 1. Suppose that values < 1.0001 are arbitrage prices. Then Property A tells us that we can find a value slightly bigger than 1.00009 that is also an arbitrage price. Suppose that values < 1.00009001 are also arbitrage prices. Continuing in this way we might *never* reach 1.0001, 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 13. 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 23: 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.

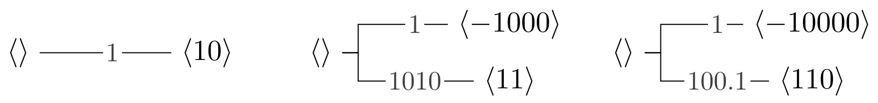
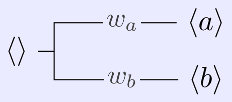


FIGURE 23. FRP kinds with the same risk-neutral price. Are you indifferent to which of these payoffs you get?

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 28. Assuming $a < b$, can the risk-neutral price of the FRP with kind



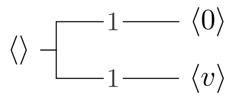
be less than a? Greater than b? Why or why not?

Can it be equal to a or b? (Remember $w_a, w_b > 0$.) Why or why not?

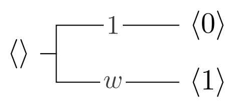
If w_b is very much bigger than w_a, do you expect the risk-neutral price to be closer to a or to b?

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:

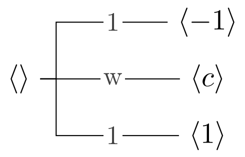
1. For various values of v,



2. For various values of w,



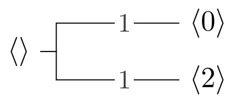
3. For various values of $-1 < c < 1$ and of w, starting with $w = 1$,



Try some other examples as well. Can you guess the relationship between an FRPs 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 tell you about the risk-neutral price?

8.1 Fundamental Properties of Risk-Neutral Prices

The risk-neutral price of an FRP X represents a good prediction of X's value. This might seem like an odd statement at first. For instance, we saw the FRP with kind



has a risk-neutral price of 1 – but 1 is *not a possible value* of X. So if you predict 1, you will aways be wrong. But the market captures a different notion of prediction: prediction *in the aggregate*. 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 an optimal prediction for that value, in a particular sense to be described below.

And 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: at any price $c < v$, we make $\$v - c > 0$ for every unit purchased.

If the *smallest possible value* of X is a, then any $c < a$ must be an arbitrage price, because X's value will certainly be $> c$. Hence, X's risk-neutral price is $\geq a$.

For a real-number s, 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. Consider a large batch of FRPs with the same kind as X: $X_{[1]}, \ldots, X_{[m]}$. At the market, we can choose to purchase the batch $sX_{[1]}, \ldots, sX_{[m]}$. If c is any arbitrage price for X, then for large enough m, the batch $X_{[1]}, \ldots, X_{[m]}$ would essentially guarantee us a profit. If $s > 0$, then we would also get a profit from the same batch with payoffs $sX_{[1]}, \ldots, xX_{[m]}$, because all the payoffs are just scaled by s, so sc is an arbitrage price for sX. Hence, the risk-neutral price of sX is just s times the risk-neutral price for X. (If $s < 0$, we just use the same argument scaling by $-s$, yielding the same result.)

Take $s = -1$ and apply the previous two paragraphs. If b is the *largest possible value* of X, then $-b$ is the smallest possible value of $-X$. So the risk-neutral price of $-X$ is $\geq -b$, meaning that the risk-neutral price of X is $\leq b$.

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 `E(X)`.)

Consider one more property of risk-neutral prices. Let $d = \mathsf{dimension}(X)$ and let Y and Z be any two components of X. That is, $Y = X_i$ and $Z = X_j$ for some $1 \leq i, j \leq d$. Now define a statistic ψ by $\psi(x_1, x_2, \ldots, x_d) = x_i + x_j$ and write $\psi(X)$ as $X_i + X_j$ or, equivalently, $Y + Z$.

Consider the three FRPs $Y + Z$, Y, and Z. We can find $\mathbb{E}(Y + Z)$ from $\mathbb{E}(Y)$ and $\mathbb{E}(Z)$. If $c_1 \leq \mathbb{E}(Y)$ and $c_2 < \mathbb{E}(Z)$, then $c_1 + c_2$ is an arbitrage price for $Y + Z$: 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 the last section

$$\mathsf{Sum}(a :: b) = \mathsf{Sum}\left(\langle\, \mathsf{Sum}(a), \mathsf{Sum}(b)\,\rangle\right),$$

this generalizes to any number of components.

We just derived the following properties of $\mathbb{E}(X)$ for an FRP X.

Box 14. Key Properties of Risk-Neutral Prices. Let X be an FRP.

Constancy

If X is a constant – has only one possible value v – then

$$\mathbb{E}(X) = v. \tag{8.1}$$

Scaling

If s is a real number and sX is the transformed FRP scaling X by s, then,

$$\mathbb{E}(sX) = s\,\mathbb{E}(X). \tag{8.2}$$

Ordering

If all possible values of X are $\geq a$ and $\leq b$, then

$$a \leq \mathbb{E}(X) \leq b \tag{8.3}$$

Additivity

If $X_1, X_2, \ldots, X_n$ are the components of X, then

$$\mathbb{E}(X_1 + X_2 + \cdots + X_n) = \mathbb{E}(X_1) + \mathbb{E}(X_2) + \cdots + \mathbb{E}(X_n). \tag{8.4}$$

In terms of the Sum statistic and equation 8.6 below, we can also write this as $\mathbb{E}(\mathsf{Sum}(X)) = \mathsf{Sum}(\mathbb{E}(X))$.

Example 8.1. If X has kind described by `(<> 1 <-1> 4 <1>)`, what is $\mathbb{E}(X^2)$?

(Recall that X^2 denotes the transformed FRPs by statistics $\psi(x) = x^2$; see Definition 7.)

X^2 is a transform of X. When we see the value v that X produces at its output ports, we feed that value through a statistics adapter that passes v^2 to the next stage. We don't know X's value, but we know that it will be either -1 or 1. Thus the possible values of X^2 is the set of v^2 for v either -1 or 1. This means that 1 is the only possible value of X^2. It is constant.

Hence, by the Constancy property (8.1), $\mathbb{E}(X^2) = 1$.

(Note: for the remainder of this section, we use the shortened `pg>` and `..>` for the playground prompts so that long lines fit better on the page.)

Example 8.2. In the previous section, Alice and Bob figured out a clever way to compute `Sum(d6 ** 100)` in the playground. This is the kind of an FRP that represents the roll of 100 balanced 6-sided dice. To compute the risk-neutral price for this, we do not need such clever tricks.

```
pg> d6 = uniform(irange(1, 6))
pg> D_6 = frp(d6)
pg> D_6
An FRP with value <4>
```

This creates an FRP `D_6` with kind `d6` that simulates a single roll of a six-sided die. We can create FRPs to simulate the rolls of 4 or 8 dice similarly, but `d6` $\star\star$ 4 and `d6` $\star\star$ 8 have sizes 1296 and 1,679,616, which are doable but getting large. Instead, we can create the FRPs directly

```
pg> rolls4 = clone(D_6) * clone(D_6) * clone(D_6) * clone(D_6)
An FRP with value <2, 1, 3, 2>
```

The value of `rolls4` shows us the four rolls; it has been computed directly by activating each of the four clones and plugging their outputs into `roll4`'s input port. The kind of the FRP has not yet been computed, as the message indicates.

This allows us to construct FRPs even if the kind is hard compute, though in this case we could just call `kind(rolls4)` to find it. (Try it!)
Note that when we use the independent mixture operator on FRPs, we *need* to use the `clone` function explicitly if we want distinct FRPs for each term. Because each FRP has only one value fixed for all time, the following would not be quite what we wanted:

```
pg> D_6 * D_6 * D_6 * D_6    # not quite right
An FRP with value <4, 4, 4, 4>.
```

No matter how many times you do this, you will get the same value each time.

Now to simulate more rolls, we could just type `clone(D_6) * clone(D_6) * ...` many times, but that is tedious. Instead, in the playground, independent mixture powers of FRPs, like `D_6 ** n`, are treated as a shorthand for the expression `clone(D_6) * ... * clone(D_6)` with `clone(D_6)` repeated n times. (The more literal meaning without the `clone`s would only rarely be of interest.)

```
pg> D_6 ** 8
An FRP with value <5, 6, 3, 4, 4, 1, 4, 3>. (It may be slow to evaluate its kind.)
pg> rolls100 = D_6 ** 100
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.)
pg> Sum(rolls100)
An FRP with value <322>. (It may be slow to evaluate its kind.)
pg> E(Sum(rolls100)) = 350
```

The risk-neutral price for for `roll100` is computed using the properties in Box 14 and Definition 15.

First, remember that `E(D_6) == E(clone(D_6))` because the risk-neutral price for an FRP only depends on its kind not its particular value. And we know that

```
pg> E(D_6)
7/2
```

Second, Definition 15 tells us that `E(D_6 ** 100)` is equal to `<E(clone(D_6)), E(clone(D_6)), ..., E(clone(D_6))>`, whose value we know. Indeed, we can compute

```
pg> E(D_6 ** 100)
<7/2, 7/2, 7/2, ..., 7/2, 7/2>
```

where I've excised 95 of the values for brevity. Finally, Additivity (equation 8.4) tells us that

```
pg> E(Sum(D_6 ** 100)) == Sum(E(D_6 ** 100))
True
pg> E(Sum(D_6 ** 100))
350
```

as we saw earlier.

Example 8.3. Here we revisit Alice's deck of cards from the previous section. Let S be the FRP of dimension 52 (and size 52!) whose values are all permutations of $1, 2, \ldots, 52$. Let $\mathcal{A} = \{1, 14, 27, 40\}$ be a set of four special values in $[1\,..\,52]$.

We can use the FRP D 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 value in $\mathcal{A}$; a_2 is the number of components strictly between the first two values in $\mathcal{A}$; a_3 is the number of components strictly between the second and third value in $\mathcal{A}$; a_4 is the number of components strictly between the third and fourth value in $\mathcal{A}$; and a_5 is the number of components strictly *after* the fourth value in $\mathcal{A}$. These quantities model the number of cards before the first ace, between the first and second aces, and so forth.

Puzzle 29. Show how to define the statistic ψ in the playground. Call it `between_aces`.

We can use the `shuffle` FRP factory to create S:

```
pg> S = shuffle(irange(52))   # Values are permutations of 1..52
pg> 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.

```
pg> A = between_aces(S)
pg> 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

```
pg> A[1]
An FRP with value <11>. (It may be slow to evaluate its kind.)
pg> A[3]
An FRP with value <27>. (It may be slow to evaluate its kind.)
```

and similarly for the others.

To start, we would like to compute $\mathbb{E}(A_1 + A_2 + A_3 + A_4 + A_5)$. The FRP here is a statistic applied to A, which is in turn a statistic applied to S. This statistic can be expressed in multiple equivalent forms.

```
pg> Sum(A)
pg> Sum(between_aces(S))
pg> S ^ Sum(between_aces)
pg> S ^ between_aces ^ Sum
```

These are all the same FRP. (Make sure you understand why.)

Puzzle 30. 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:

```
pg> psi = Sum(between_aces)
pg> FRP.sample(100, psi(S))
```

What do the results tell you? Can you explain this table? What happens if you demo 1000 samples?

What we discovered in the previous puzzle is an invariant that lets us find the risk-neutral price. Define $\zeta = \mathsf{Sum} \circ \psi$, "Sum after `between_aces`." Then, $\zeta(S) = A_1 + A_2 + A_3 + A_4 + A_5$. By property (8.1), the invariant tells us that we have that $\mathbb{E}(\zeta(S)) = 48$. This result follows from logic as much as computation, but the computations make the result clear and concrete.

With the Additivity property for risk-neutral prices, we have

$$\begin{aligned} 48 &= \mathbb{E}(A_1 + A_2 + A_3 + A_4 + A_5) \\ &= \mathbb{E}(A_1) + \mathbb{E}(A_2) + \mathbb{E}(A_3) + \mathbb{E}(A_4) + \mathbb{E}(A_5), \end{aligned}$$

so we've learned something about the risk-netural prices of the individual components.

With a bit more logic, we can go even further. The key idea is to exploit the *symmetry* of the system. Recall Alice's and Bob's observation from the last section that a shuffle of the deck does not change the kind of S. That is, if ϕ is *any permutation statistic* that applies a fixed permutation to the components of the 52-tuples produced by S, we have

$$\mathsf{kind}(\phi(S)) = \mathsf{kind}(S).$$

We can then ask what this symmetry implies about the kinds of FRPs obtained by transforming S, such as $\zeta(S)$.

We can try it. But to illustrate the ideas, it will be easier to work with a smaller "deck" and a simpler but analogous statistic. Let's look at all shuffles of $[1\,..\,5]$ and instead of `between_aces`, let's consider the analogous `two_four_gaps`, that uses $\{2, 4\}$ instead of the set $\mathcal{A}$.

```
pg> T = shuffle(irange(5))
pg> kind(T)
```

I'm not showing the tree here, but you should look at it. It's still small enough to understand and examine.

First, let's confirm Alice's and Bob's observation. Pick an arbitrary permutation and turn it into a statistic. For instance, this will work:

```
pg> phi = Permute(clone(T).value)  # This is a statistic
pg> kind( phi(T) )
```

Look at the tree closely; it's in canonical form and the two are the same. Indeed, we can check this explicitly in the playground:

`T.kind` and `kind(T)` are equivalent; we typically use the one that is easier to read and understand.

```
pg> Kind.compare(T.kind, phi(T).kind)
The two kinds are the same within numerical precision.
```

The permutation was arbitrary, and the results will be the same – for the reasons Alice and Bob gave – if we try it with any other permutation.

Now, let's see the impact of a permutation on our analogue of `between_aces`. The statistic `two_four_gaps` can be defined as follows; it has dimension 5 and co-dimension 3. It scans through the "deck" looking for 2's and 4's and computing the gap between successive such cards (and the beginning/end of the deck).

```
@statistic(name='A simpler version of between_aces', dim=5, codim=3):
def two_four_gaps(deck):
    targets = {2, 4}  # These *are* the cards we're looking for
    diffs = []
    last_index = -1
    for i, card in enumerate(deck):
        if card in targets:
            diffs.append(i - last_index - 1)
            last_index = i
    diffs.append(len(deck) - last_index - 1)
    return diffs
```

We can compare the kinds of T transformed by this statistic, before and after an extra permutation.

```
pg> kind( two_four_gaps(T) )
pg> kind( two_four_gaps(phi(T)) )  # equivalent to kind(T ^ phi ^ two_four_gaps)
```

Looking at the kinds, we see they are the same! (You can also use `Kind.compare`.)

So, the kind of `two_four_gaps(T)` does not change under permutations. What about the components?

```
pg> kind( T ^ two_four_gaps ^ Proj[1] )
pg> kind( phi(T) ^ two_four_gaps ^ Proj[1] )
```

again have the same kind. See Property 16 below.

Let's call the components of `two_four_gaps(T)` $\langle G_1, G_2, G_3\rangle$. What we just saw implies that the kind of G_1 is invariant under permutations of the deck, and similarly for G_2 and G_3. And yet if I reverse the deck – which is just a permutation – I just swap the values of G_1 and G_3. (Why?) Extra shuffles of the deck change the values of these components but do not change their kinds. This suggests that G_1, G_2, and G_3 *have the same kind*. And we can check this:

```
pg> G = T ^ two_four_gaps
pg> kind( G[1] )
```

```
    pg> kind( G[2] )
    pg> kind( G[3] )
```

All three are the same, as we guessed. So

```
    pg> E(G[1]) == E(G[2]) and E(G[1]) == E(G[3]) and E(G[2]) == E(G[3])
    True
    pg> E(Sum(G))
    3
    pg> E(Sum(G)) == E(G[1]) + E(G[2]) + E(G[3])
    True
```

All the $\mathbb{E}(G_i)$'s are equal and they sum to 3, so $\mathbb{E}(G_i) = 1$.

The exact same phenomenon occurs with S and A in place of T and G. It's just more items with the same logic. We saw above by the Constancy property that

$$\mathbb{E}(A_1 + A_2 + A_3 + A_4 + A_5) = 48.$$

And using the symmetry as we did with T, we see that all 5 A_k's *have the same kind*. So, their risk-neutral prices are also the same. By the Additivity property (8.4) we have

$$\begin{aligned}\mathbb{E}(A_1 + A_2 + A_3 + A_4 + A_5) &= \mathbb{E}(A_1) + \mathbb{E}(A_2) + \mathbb{E}(A_3) + \mathbb{E}(A_4) + \mathbb{E}(A_5)\\ &= 5\mathbb{E}(A_1)\end{aligned}$$

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 risk-neutral prices of A_i.

This example shows us the power of the properties we've discovered about risk-neutral prices. They let us compute the risk-neutral prices without explicitly considering every possible value. This also shows that the risk-neutral prices are often easier to compute than the full kinds.

Example 8.4. 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 risk-neutral prices for this kind and in the process discover a useful general property of risk-neutral prices in the case where the two statistics are related.

As a concrete example, consider

```
pg> X = frp(either(0,1) >> {0: uniform(1, 2, 3) * either(4, 5),
..>                         1: either(7, 2, 1/7) * either(4, 5)})
pg> X
An FRP of dimension 3 and size 10 with value <0, 3, 5>
pg> 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>
pg> psi = Max
pg> phi = Max - Min
```

Suppose I want the risk neutral price for `phi(X)` having observed the fact that `psi(X) < 9`. In the playground, we would like to write this as `E(phi(X) | (phi < 9))` but that is not *quite* right because this passes the values of the transformed kind `phi(X)` to the condition instead of the value of `X` itself. We can express this, correctly, as

```
E( phi(X | (psi < 9)) )
```

and this is fine. However, we would like the playground notation to match our conceptual/mathematical notation – $\mathbb{E}(\phi(X) \mid \psi(X) < 9)$ – a bit better. For this, we use some "syntactic sugar", with `phi@X` in place of `phi(X)`:

```
pg> E(phi@X | (psi < 9))
14/3
```

Here, think of `phi@X` (read "phi at X") as meaning the same thing as `phi(X)`; the only difference is that it passes the value of `X` itself to the condition. This notation works with kinds too, as we will see.

Now, we want to find `E(phi@X | (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) ).
```

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 prices 4, 5, 6, and 8 when a is 4, 5, 7, and 9, respectively.

Rather than just computing the risk-neutral prices, it makes sense to consider the associated kinds. In the playground, we can do

```
pg> 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):

```
pg> 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:

```
pg> f = E(range_given_max)
```

which returns a *function* from `a` to the risk neutral price we want.

```
pg> [f(a) for a in [4, 5, 7, 9]]
[4, 5, 6, 8]
```

Nice.

One more key point. The statistic `phi` here has a special form. If we define `zeta(x, y) := y - Min(x)`, then

The `:=` denotes definition.

```
phi(x) == zeta(x, psi(x)) == psi(x) - Min(x).
```

If we *know* that `psi(x)` has a particular value `a`, then `phi(x) = a - Min(x)`. So making predictions about `phi(X)` with that knowledge is equivalent to making predictins about `a - Min(X)` with that knowledge. Put another way: if I were making predictions about the value of `phi(X)` and you were making predictions about the value of `a - Min(X)` and both of us had the information that `psi(X) == a`, we would make the *same predictions.*

This is one way to state the ***substitution property*** for risk-neutral prices. You can check it in the playground. For instance, compute both `E( (4 - Min)@X | (psi == 4) )` and `E( phi@X | (psi == 4) )` and compare them, and similarly for the other values of `psi(X)`.

This yields the ***substitution rule***: if `psi` and `phi` are statistics compatible with `X`, and if `phi(x)` has the form `zeta(x, psi(x))`, then

```
E( phi@X | (psi == a) ) == E( zeta(__, a)@X | (psi == a) ).
```

Given the knowledge that `psi(X)` has the value `a`, we can simply substitute `a` for `psi(x)` wherever we see it to the left of the `|`.

Mathematically, this looks like

$$\mathbb{E}(\zeta(X, \psi(X)) \mid \psi(X) = a) = \mathbb{E}(\zeta(X, a) \mid \psi(X) = a). \tag{8.5}$$

We will see this in more data later and see why it is useful. In short, it lets us use given information to *simplify* the quantities we want to predict.

We have been discussing risk-neutral prices for scalar FRPs, but we can extend our definition to higher dimensions, and all the properties above continue to hold.

Definition 15. If FRP X has dimension n and FRPs $\langle X_1, X_2, \ldots, X_n \rangle$ are its scalar components, then we define

$$\mathbb{E}(X) = \langle \mathbb{E}(X_1), \mathbb{E}(X_2), \ldots, \mathbb{E}(X_n) \rangle, \tag{8.6}$$

that is, the list of risk-neutral prices of the components.

We can think of non-scalar FRP as offering a *portfolio* of payoffs.

From our thinking about risk-neutral prices and our empirical studies of them, it is clear that the risk-neutral price of an FRP is *determined by its* ***kind*** not by its particular value. So all FRPs with the same kind have the same risk-neutral price.

In fact, this goes the other way as well. The kind of an FRP X is determined by the risk-neutral prices *for all transformed FRPs* derived from X. This yields an important result:

Property 16. Two FRPs X and Y have the *same kind* if and only if they have the same set of possible values and

$$\mathbb{E}(\psi(X)) = \mathbb{E}(\psi(Y)) \tag{8.7}$$

for *every compatible statistic* ψ.

This tells us that, *in aggregate*, the risk-neutral prices 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 16 says that two FRPs with the same kind yield equal risk-neutral prices when transformed by the same statistic. The other direction *seems* less useful because it requires that the risk-neutral prices of transformed FRPs be the same for all compatible statistics. But it turns out that this direction works with smaller collections of statistics if they are sufficiently rich. In other words, we can "determine" the kind of an FRP with the risk-neutral prices of a well-chosen collection of statistics.

Definition 17. A collection Ψ of statistics compatible with an FRP X ***determines*** the kind of X if for any FRP Y with $\mathsf{values}(X) = \mathsf{values}(Y)$:

$$\mathbb{E}(\psi(X)) = \mathbb{E}(\psi(Y)) \text{ for all } \psi \in \Psi \text{ implies } \mathsf{kind}(Y) = \mathsf{kind}(X). \tag{8.8}$$

Puzzle 31. If Z is an FRP of dimension 2, the collection $\mathsf{proj}_1, \mathsf{proj}_2$ 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 $\mathbb{E}(\mathsf{proj}_1(Z) = \mathbb{E}(\mathsf{proj}_1(U))$ and $\mathbb{E}(\mathsf{proj}_2(Z) = \mathbb{E}(\mathsf{proj}_2(U))$ but Z and U have *different kinds*.

There are useful collections that do determine the kind. Suppose X is a scalar FRP, and consider the collection of statistics ψ_a where $-\infty < a < \infty$ is a real number and $\psi_a(x) = 1$ if $x \le a$ and $\psi_a(x) = 0$ if $x > a$.

We want to see that if Y has the same possible values as X and $\mathbb{E}(\psi_a(X)) = \mathbb{E}(\psi_a(Y))$ for *every* a, then X and Y must have the same kind.

We can sketch out the argument in the playground. Define any scalar FRP X and write $\psi_a(x)$ as `psi(a)(x)` where `phi(a)` is a statistic.

```
pg> def psi(a):
..>     @scalar_statistic(dim=1)           # psi(a) == le_a is a scalar statistic
..>     def le_a(x):
..>         return int(x <= a)             # returns 0 or 1
```

```
pg> vs = sorted(values(X, scalarize=True))  # Get X's values in order as scalars
pg> prev = min(vs) - 1                      # A value *below* X's minimum value
pg> stats = {}                              # We will build a dictionary of statistics
pg> for value in vs:
..>     a = 0.5*(value + prev)              # a is halfway between value and prev
..>     stats.append[value] = psi(value) - psi(a)
..>     prev = value
```

Now, `stats` contains all the statistics $\psi_v - \psi_a$ where v is a possible value of X and where there are no other values of X between a and v.

Now from this, find the kinds

```
pg> ks = {value: stat(kind(X)) for value, stat in stats.items}
```

and picking any value `v`, *look at the kind*:

```
pg> ks[v]
```

What does it tell you? What is its risk-neutral price? If Y were an FRP with the same values as X and you did the same calculation, is it possible for the risk-neutral prices of these kinds to be the same but the kinds different?

Compute them all

```
pg> Eks = {value: E(ks[value]) for value in ks}
```

What are these prices and how do they relate to the kinds?

This calculation shows that if these prices are the same, then X and Y must have the same kind, and all our predictions about X are the same our predictions about Y.

That's an important argument, and it's worth going over *in the playground* to make sure its clear.

It will be conceptually powerful to recast these quantities a bit, using the `D_` operator from the playground. This operator is a bit mind bending at first, so we will just introduce it lightly right now.

Property 16 tells us that the risk-neutral prices $\mathbb{E}(\psi(X))$, varying over compatible statistics ψ, determine the kind of X. We have seen that $\mathbb{E}(\psi(X))$ is our best prediction of the value of $\psi(X)$. So what this means is that our predictions of the values of transformed FRPs embody our knowledge of the original FRP.

The `D_` operator takes an FRP (or kind) as input and returns a function mapping every (compatible) statistic to its corresponding prediction. The function `\D_(X)` embodies all our knowledge of X's value. It is defined by

```
D_(X)(psi) = E(psi(X)).
```

> **Example 8.5**. Try the following, computing the risk-neutral prices as shown and directly.
>
> ```
> pg> X = frp(uniform(1, 2, 3) * uniform(1, 2, 3))
> pg> f = D_(X)
> pg> f(Sum)
> 4
> pg> f(Max)
> 22/9
> pg> f(Min)
> 14/9
> pg> f(Cos)
> -0.28861
> pg> f(Sin)
> 0.630629
> ```

The importance of `D_` is that if you have `D_(X)`, you have the same information as in `kind(X)`. And in particular, you can make good predictions about any feature of X's value. We will come back to this idea again.

8.2 Computing Risk-Neutral Prices

So let's take stock. We have

1. A precise definition of risk-neutral prices that captures our "best" prediction of an FRPs value.

2. A simple notation for the risk-neutral price of an FRP that extends to any dimension.

3. A set of key properties that risk-neutral prices must follow for any FRP derived from the logic of the definition.

4. A result that the kind of an FRP determines its risk-neutral price and is in turn determined by the risk-neutral price of every transformed FRP.

These are powerful already, and we have seen how we can already use this to compute predictions. But it would be nice if there were an easier way to express $\mathbb{E}(X)$ for an FRP/kind.

The good news is that there is, and that we can find it by using our empirical investigations. Let's look at a sample demo to motivate the argument.

```
market> 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%)
market> 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 arbitrag 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. The risk-neutral price for this kind is 1.8.

Definition 18. If X is an FRP of size m with values $v_1, \ldots, v_m$ and corresponding *canonical* weights $p_1, \ldots, p_m$, then the risk-neutral price of X is given by

$$\mathbb{E}(X) = p_1 v_1 + \cdots + p_m v_m. \tag{8.9}$$

Risk-neutral prices are thus *weighted averages* of the FRP's values.

Two important points about this definition:

1. It works for *any dimension* FRP. What we need here is that we can weight and add the values, which are lists of numbers of the same dimension. And indeed we can by defining for any real number c:

$$c\langle x_1, x_2, \ldots, x_d\rangle = \langle cx_1, cx_2, \ldots, cx_d\rangle \tag{8.10}$$

$$\langle x_1, x_2, \ldots, x_d\rangle + \langle y_1, y_2, \ldots, y_d\rangle = \langle x_1 + y_1, x_2 + y_2, \ldots, x_d + y_d\rangle. \tag{8.11}$$

   With this, equation 8.9 makes sense and is consistent with equation 8.6.

2. If the kind(X) is in compact but not canonical form, then equation 8.9 becomes

$$\mathbb{E}(X) = \frac{w_1 v_1 + \cdots + w_d v_d}{w_1 + \cdots + w_d}. \tag{8.12}$$

   We use p_i's to indicate canonical weights (that sum to 1) and w_i's when this need not be true.

Example 8.6. If X has kind described by `(<> 1 <-1> 4 <1>)`, what is $\mathbb{E}(X)$, $\mathbb{E}(X^3)$, and $\mathbb{E}(X \star X)$?

(Recall that X^2 and X^3 denote the transformed FRPs by statistics $\psi(x) = x^2$ and $\phi(x) = x^3$, respectively. See Definition 7.)

Applying equation 8.9 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.

The last result in this example illustrates another common pattern. If $X_1, X_2, \ldots, X_n$ are scalar FRPs, then $X = X_1 \star X_2 \star \cdots \star X_n$ is an n-dimensional FRP with scalar components $\langle X_1, X_2, \ldots, X_n\rangle$. This follows from the properties of an *independent mixture*. So equation 8.6 tells us that

$$\mathbb{E}(X_1 \star X_2 \star \cdots \star X_n) = \langle \mathbb{E}(X_1), \mathbb{E}(X_2), \ldots, \mathbb{E}(X_n)\rangle \tag{8.13}$$

Example 8.7.

In the revisited disease testing example on page 72, we computed the kind of an FRP whose outcome indicates whether someone has the disease when it is known that they test positive.

```
pg> has_disease_pos = (dStatus_and_tResult | (Test_Result == 1))[Disease_Status]
    ,---- 999/1094 ---- 0    # No disease
<> -|
    `---- 95/1094  ---- 1    # Yes disease
```

If D_+ is an FRP with kind `has_disease_pos`, what is $\mathbb{E}(D_+)$?

$$\mathbb{E}(D_+) = \frac{999}{1094}\cdot 0 + \frac{95}{1094}\cdot 1 = \frac{95}{1094} \approx 0.0868.$$

This illustrates another common pattern. An FRP whose only values are 0 and 1 acts as an indicator of whether something happens. (Here 0 is playing the role of false or $\bot$, and 1 is playing the role of true or $\top$.) We will call such an FRP an *event*.

Equation 8.9 shows that for any event I, $\mathbb{E}(I)$ is just the canonical weight on its 1 branch. We will 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.

The discussion just after Puzzle 31 also made good use of this fact.

Example 8.8. Try these calculations; look at the weighted averages that you get; and compare them to the risk-neutral prices that we compute. I've omitted the results here.

```
pg> coin = either(0, 1)  # Model: 1 for heads, 0 for tails; equally weighted
pg> flips10 = coin ** 10
pg> num_heads_in_10 = Sum(flips10)
pg> num_heads_in_10
pg> E(num_heads_in_10)
pg> 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 revenue neutral price. Notice how the Properties we discovered earlier help us compute the last value without actually hitting enter.

For a value like `E(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:

```
pg> coin ** 4
pg> Sum(coin ** 4)
pg> 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 8.9.

Let's look at another interesting transform along with some observed information in a conditional.

```
pg> @scalar_statistic('Index of first heads, or 1000')
..> def first_heads(x):
..>     return 1 + index_of(1, x, not_found=999)
pg> flips16 = coin ** 16
pg> when_heads = first_heads(flips16)
```

We start by defining a statistic that gives the index of the first head in 16 flips, or (arbitrarily) 1000 if a head did not occur.

Compare the following:

```
pg> when_heads
pg> when_heads ^ IfThenElse(__ > 10, 1000, __)
pg> when_heads ^ IfThenElse(__ > 10, 1000, __)
pg> (when_heads | (__ > 4)) ^ IfThenElse(__ == 1000, __, __ - 5)
```

What do these kinds mean? How do they compare? Do you think this is a coincidence?

The second is just the kind of `when_heads`, but we map every value bigger than 10 to the arbitrary value of 1000. We do this to ease comparison with the third. The third is the kind of an FRP that gives the number of 1's (heads) *after* the fourth flip when you have observed no 1's (heads) in the first four flips. The statistic at the end simply shifts the values back to the scale of the first two kinds.

8.3 Expectations and Probabilities

We have learned a lot about risk-neutral prices by reasoning from the definition, and we have derived several ways to find the risk-neutral price for an FRP. As we proceed, we will see several different interpretations of these prices, but the critical theme is that having these prices gives us an effective *prediction* for the value of that FRP (and any with the same kind). This focus on prediction underlies may of the key ideas to come, so here we assign a more general name to the risk-neutral price that emphasizes this predictive interpretation.

Definition 19. If X is an FRP, we call its risk-neutral price the ***expectation*** of X, denoted by $\mathbb{E}(\boldsymbol{X})$.

If in addition, c is a condition on X, then the risk-neutral price of X given that c is true is denoted by $\mathbb{E}(X \mid c)$. When $c = \top$, the predicate that is always true, we have $\mathbb{E}(X) = \mathbb{E}(X \mid \top)$.

As we have seen, if two FRPs X and Y have the same kind, then our predictions $\mathbb{E}(\psi(X))$ and $\mathbb{E}(\psi(Y))$ are equal for any compatible statistic ψ. So the expectation is really a *property of the kind*. We thus can talk about the expectation of a kind in the same way, though in practice we tend to focus on the FRPs as those are the quantities that we want to predict.

Recall that when we apply a conditional on some condition c, we get a new FRP/kind, $X \mid c$, with its own expectation $\mathbb{E}(X \mid c)$. These are often referred to as *conditional expectations*, but really they are just plain old expectations in the context of additional information: if $c = \top$, i.e., it is always true, then $\mathbb{E}(X \mid c) = \mathbb{E}(X)$. A big part of our analysis will be using new information to update our predictions – with conditionals.

Finally, in practice, we are frequently interested in whether or not some observable outcome occurs. Did it occur or not. We model such binaries with scalar FRPs that have possible values 0 (for false) and 1 (for true).

Definition 20. A scalar FRP with possible values 0 and 1 is called an ***event***. The event is said to have *occurred* if the FRP produces the value 1.

Puzzle 32. If V is an event and ψ is the statistic $\psi(v) = 1 - v$, then what can you say about the FRP $\psi(V)$?

Because we often consider events, it is natural that we would want to predict whether the event will occur. And just like with any FRP its the risk-neutral price – that is, its *expectation* – gives a prediction. But remember our analysis earlier. If an FRPV has a biggest possible value b, then $\mathbb{E}(V) \le b$; if it has a smallest possible value a, then $a \le \mathbb{E}(V)$.

So if V is an event, $0 \le \mathbb{E}(V) \le 1$. When $\mathbb{E}(V)$ is near 1, our risk-neutral price is close to 1, meaning that we predict that V will tend on average to be close to 1. Similarly, when $\mathbb{E}(V)$ is near 0, we predict that V will tend on average to be close to 0. Think about the summary tables when you run demos in the market! Thus, for events V, $\mathbb{E}(V)$ is a number between 0 and 1 that measures in some sense our confidence that the event V will occur. Because we use it so often, such expectations have a name.

Definition 21. If V is an event, then its expectation is a number between 0 and 1; we call it the ***probability*** of the event occurring.

Probabilities are nothing new. They are just expectations – risk-neutral prices – for FRPs whose values are in a particular range. So probabilities *inherit* all the properties of expectations.

Now that we are familiar with the FRPs and kinds and the various ways to use them, it's time to take them out for a spin and see what we can do with them.

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.

9 Making Predictions

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 a Statistic
- Building with a Mixture
- Constraining with a Conditional
- Computing with an Expectation

Several patterns in the use of these four operations arise frequently: marginalization (projecting onto selected components), condition*ing* (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. These problems illustrate the essential techniques of probability theory and some of the common patterns of analysis. Focus here on identifying the basic operations we have developed – mixture, transformation, conditionals, expectation. All of the complex analyses we do are built from these operations, so it is a good idea to study how they are combined to answer questions. 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, I omit the prompts (like `playground>`) from the display to save space. 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 *`".

9.1 The Big 3+1!

We have four basic operations – transforming with a statistic, constraining with a conditional, building with a mixture, computing with an expectation – that we perform on FRPs/kinds, and each serves a particular need.

1. **Transforming with a Statistic**.

   We use statistics to express and **answer questions**. When an FRP produces a value (or a kind has values) that you want to extract some detail or summary or feature from, you apply a statistic.

   Do we win the game? What is the area of the random polygon? How long is Theseus's path in the maze before he escapes? All of these questions are asked by applying a data-processing algorithm to a richer value, e.g., the game state, the coordinates of the polygon's vertices, Theseus's path.

   A statistic is a function that maps values (i.e., tuples of numbers) to values. That's all it does, but we can use that function to *transform* an FRP or 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 combine branches that map to the same value in the transformed tree, adding their weights.

2. **Constraining with a Conditional**.

   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.

3. **Building with Mixture**.

   We use mixtures to break a complex process into simpler stages and to and **to build a model of the process by combining those stages**. When a system can most easily be described by looking at parts in isolation and combining them, think about using a mixture.

   If the different stages of the process do not interact – that is, if what happens in one stage does not influence what happens in another – then we use an independent mixture. In an independent mixture, all possible outcomes of the different stages are combined.

   More generally, though, the different stages will interact. What happens at one stage will influence the outcomes of later stages. A mixture reflects this by passing the values produced at one stage into the next and collecting that entire history.

   A conditional kind takes values from the earlier stages and produces the *kind* for the next stage. A conditional FRP takes the value of the earlier stages and produces the *FRP* that gives the outcome of the next stage. The are conditional in the sense that what happens next is contingent on what came before.

4. **Computing with an Expectation**.

   We use expectation **to compute our best prediction of an FRP's value**. The expectation represents the risk-neutral price of the FRP and so reflects a "typical" value that is "close" to what the FRP will produce.

   From their definition as risk-neutral prices, expectations inherit many useful properties, additivity, scaling, ordering, constancy, and more. From these properties, we deduced that expectations are *weighted averages*. The expectation of a kind is the weighted average of the kinds' values using the weights for each value. The expectation of an FRP is just the expectation of 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. If the FRP is an event, then the expectation is interpreted as a probability, a measure of our certainty that the event will occur.

All the analyses and techniques that we use in probability theory are built on these big 3+1. For example:

- *Marginalization* is the operation of extracting out selected components a value. This is just transformation by a projection statistic.
- *Conditioning*, aka the Method of Hypotheticals, averages the outputs of a conditional kind using another kind's weights. This is just mixture followed by marginalization.
- Bayes's Rule allows us to "reverse the conditionals" and learn about an earlier stage of a process by viewing the outputs of a later stage. This is mixture followed by a conditional followed by marginalization.

Statistical inference combines all four operations. We study Markov chains, a special kind of random process, through conditioning. Asymptotic analysis looks at transformations by statistics of very large mixtures, where each component has some regular properties. The list goes on and on.

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. It's turtles all the way down!

9.2 Playground Types

The examples in the following subsections make heavy use of the FRP playground to show both how we *model* the situation at hand and how we use the big 3+1 to analyze that model.

To make that easier, it's worth focusing on what types of things the different entities in the playground are and what types of things the different combinators expect. Here, we summarize the types of the entities and operations in the playground. We write $a \to b$ to indicate a function that takes as input data of type a and returns as output data of type b. For a binary operator `op`, we write $a\texttt{op}b \to c$ 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.

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.
- A `Kind` is a tree in canonical form with weights that are positive quantities and leaf nodes of type `Value`, with all leaves distinct and having the same dimension.
- An `FRP` is a device that produces a single `Value` when activated, keeping that same value for all time.

From these basic types, we define three "function" types that take an input of one type and produce an output of another:

`Statistic`: `Value` $\to$ `Value`

`ConditionalKind`: `Value` $\to$ `Kind`

`ConditionalFRP`: `Value` $\to$ `FRP`

For instance, a statistic is a function that takes and returns a value.

Next, we have several combinators that can operate on kinds or FRPs:

- `Kind` $\star$ `Kind` $\to$ `Kind` or `FRP` $\star$ `FRP` $\to$ `FRP`
- `Kind` $\star\star$ `NaturalNumber` $\to$ `Kind` or `FRP` $\star\star$ `NaturalNumber` $\to$ `FRP`
- `Kind` $\triangleright$ `ConditionalKind` $\to$ `Kind` or `FRP` $\triangleright$ `ConditionalFRP` $\to$ `FRP`
- `Kind` $|$ `Condition` $\to$ `Kind` or `FRP` $|$ `Condition` $\to$ `FRP`
- `Kind` ^ `Statistic` $\to$ `Kind` or `FRP` ^ `Statistic` $\to$ `FRP`
- `Statistic`(`Kind`) $\to$ `Kind` or `Statistic`(`FRP`) $\to$ `FRP`
- `ConditionalKind` // `Kind` $\to$ `Kind` or `ConditionalFRP` // `FRP` $\to$ `FRP`

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 `^`), 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` + `Number` → `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.

9.3 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 capture the configuration of the system at any moment. Statistics are useful for initializing and transforming between different representations.

Doubled Cards

A deck of 100 cards is labeled $1, 2, \ldots, 100$. 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.

Let D be the event that is 1 when your second card has a number exactly twice your first card. **Find the expectation of this FRP: $\mathbb{E}(D)$.** How do we interpret this number? Next, let T be the event that *either* card is twice the other. **Find the expectation of this FRP.**

We can recognize a mixture structure here: we draw the first card from the deck and then the second from the deck that is missing the first card. We define `draw` as the kind of an FRP whose value is the pair of card numbers we drew:

```
first_card = uniform(1, 2, ..., 100)
second_card = conditional_kind({first: uniform(first_card.values - {first})
                                for first in first_card.values})
```

```
draw = first_card >> second_card
```

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; its values are pairs of card numbers. To answer our question, we use a statistic. This is common: the primary use case for statistics is to extract interesting features/summaries/details from a more complex value to answer a question we care about. Our statistic here just checks if the second component of a value is equal to twice the first component:

```
is_card_doubled = Proj[2] == 2 * Proj[1]
```

We can now apply the statistic and take expectations.

```
D_kind = is_card_doubled(draw)
E(D_kind)
FRP.sample(10_000, D_kind)
```

The answer here is 1/198. This is a probability that we interpret as meaning that in a large demo of FRPs with this kind, this event will tend to happen on about 1 out 198 samples on average. The demo computed here is close to that proportion (0.0051), though not exact because the sample is finite.

It is worth looking at a few 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(first_card.values - {first}),
                               codim=1)
```

The `lambda first` defines an unnamed 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 (not a tuple), so `first` is a number here.

Second, we can derive the same result by applying a bit of reasoning, which is always useful for simplifying our analysis and computation. In particular, we can reason that if the first card `first` is bigger than 50, it is impossible for the target event to occur; and if it is ≤ 50, there is only 1 out of the 99 remaining cards that will make the event occur. Hence, in one step

```
Proj[2](c100 >> conditional_kind(
                    lambda first:
                        either(0, 1, 98) if c <= 50 else constant(0),
                    codim=1))
```

we get the same kind as above.

Finally, there is a built-in factory in the playground that does the same job

```
is_card_doubled(ordered_samples(2, irange(1, 100)))
```

gives a kind identical to `D_kind`

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.

In the next example, we introduce a useful shorthand notation. If X is an FRP, then for any Boolean statement about X's value, like $X > 2$, we use $\{\}$ around the statement to denote the FRP that is *event* that the statement is true. For instance, $\{X > 2\}$ is the FRP that returns 1 when X's value is bigger than 2 and 0 otherwise. This is an FRP whose input port is connected to X's All output port through a statistic adapter that tests whether the statement is true.

Hunter's Success

A group of 8 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 the event that hunter i hits the target has kind

$$\begin{tikzcd} & \langle 0\rangle \\ \langle\rangle \arrow[ur, dash, "1-p_i"] \arrow[dr, dash, "p_i"'] & \\ & \langle 1\rangle \end{tikzcd}$$

Let H be the FRP whose value is 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; getting answers from this solution for different settings of the weights.

We start by assuming one expert hunter, one good hunter, and six novices.

```
hunters = [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 hunters]
```

The variable `hits` contains a list of eight kinds, with the ith (at index $i - 1$) being the kind of the event that hunter i hits the target. Since all the hunters take independent shots, the number of hits is just the sum of the independent mixture of these:

```
number_of_hits = Sum(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:

```
E(number_of_hits)
E(number_of_hits ^ (__ > 4))
E(number_of_hits ^ (__ > 2))
E(number_of_hits ^ (__ > 0))
```

giving values 2.2, 0.0104, 0.3345, and 0.9841 to four decimal places.

Next, we will mimic this analysis but more generally, letting `hits` 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).

```
hunters = [symbol('p_' + str(i)) for i in irange(1, 8)]
hits = [weighted_as(0, 1, weights=[1 - w, w]) for w in hunters]
```

```
p = symbol('p')
all_equal = substitute_with(dict(p_1=p, p_2=p, p_3=p, p_4=p,
                                 p_5=p, p_6=p, p_7=p, p_8=p))
all_50_50 = substitute_with(dict(p_1=0.5, p_2=0.5, p_3=0.5, p_4=0.5,
                                 p_5=0.5, p_6=0.5, p_7=0.5, p_8=0.5))
one_expert = substitute_with(dict(p_1=0.1, p_2=0.1, p_3=0.1, p_4=0.1,
                                  p_5=0.1, p_6=0.1, p_7=0.1, p_8=0.9))
expert_plus = substitute_with(dict(p_1=0.1, p_2=0.1, p_3=0.1, p_4=0.1,
                                   p_5=0.1, p_6=0.1, p_7=0.7, p_8=0.9))
some_elders = substitute_with(dict(p_1=0.1, p_2=0.1, p_3=0.1, p_4=0.1,
                                   p_5=0.1, p_6=0.75, p_7=0.75, p_8=0.75))
random_skill = substitute_with({str(h): random() for h in hunters})
```

The elements of hunters are "symbols" `p_1`, `p_2`, . . ., `p_8` which can have numbers substituted for them after we do the calculations. The other variables like `all_equal` hold functions that will substitute the specific numeric values for the symbols.

Now, we follow the pattern before, but here we use a loop to do the independent mixture, applying the `Sum` statistic at the end:

```
number_of_hits = hits[0]
for h in hits[1:]:
    number_of_hits = number_of_hits * h
number_of_hits = Sum(number_of_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

```
    p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_8.
```

The expected number of hits is just the sum of the probabilities that each hunter hits the target. This holds no matter what profile we have.

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 using the substitution `expert_plus` to give values to the symbols:

```
expert_plus(exp_hits)
expert_plus(gt4_hits)
expert_plus(gt2_hits)
expert_plus(gt0_hits)
```

Looking at these, we see the same answers as we got earlier.

The one profile that is different is `all_equal`, which replaces eight *different* symbol for the common symbol called 'p'. Look at

```
all_equal(exp_hits)
all_equal(gt4_hits)
all_equal(gt2_hits)
all_equal(gt0_hits)
```

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 `8 p`. Had there been n hunters instead of eight, we can surmise that the expected number of hits would be `n p`. 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).

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 these `all_50_50` profile and

```
hot_and_cold = substitute_with(dict(p_1=0.05, p_2=0.05, p_3=0.05, p_4=0.05,
                                    p_5=0.95, p_6=0.95, p_7=0.95, p_8=0.95))
```

Both these 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(dim=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 first 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; put another way, we see more "spread" in the values 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.

Six of One, Equilateral of the Other

In a regular hexagon, we choose three vertices randomly so that all subsets of size 3 are have equal weight.

Let T be the FRP whose value is 1 if connecting the three vertices forms an equilateral triangle. (Is T an event?) Let A be the FRP whose value is the area of the triangle. Assume that the maximum width of the hexagon is 2.

Find the kind of T and $\mathbb{E}(T)$, and find the kind of A and $\mathbb{E}(A)$.

Imagine that we have labeled the vertices of the hexagon $1, 2, \ldots, 6$ in the clockwise direction from one of the vertices.

Picking three distinct vertices means picking a sample of size three from the numbers $1, 2, \ldots, 6$. We can use the kind factory `without_replacement` to find the kind of such a set:

```
vertices = without_replacement(3, irange(1, 6))
```

The resulting tuple 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 between successive numbers in that list. So:

```
is_equilateral = vertices ^ (Diff == (2, 2))
E(is_equilateral)
```

yielding $\mathbb{E}(T) = 1/10$. This makes sense as there are two choices ($\langle 1, 3, 5\rangle$, $\langle 2, 4, 6\rangle$) out of twenty that produce the required condition. Note that we can define `T = frp(is_equilateral)`.

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 which takes the three side lengths of the triangle:

```
@statistic
def heron(a, b, c):
    s = (a + b + c) / 2
    return Sqrt(s * (s - a) * (s - b) * (s - c))
```

Now we need to find the side lengths of the triangle, which will come 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(vertices):
    return [vertex_dists[numeric_abs(vertices[i] - vertices[(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. We add a `clean` operation to eliminate round-off error in the calculations:

```
clean( without_replacement(3, irange(1, 6)) ^ side_lengths ^ heron )
```

which yields the kind of A

```
    ,---- 0.30 ---- 0.43301
<> -+---- 0.6 ----- 0.86603
    `---- 0.1 ----- 1.2990
```

with exact values $\sqrt{3}/4, \sqrt{3}/2, 3\sqrt{3}/4$ and expectation $\mathbb{E}(A) = 0.45\sqrt{3} \approx 0.7794$.

Tournament

Eight players are ranked and play in an elimination tournament. In the first round, player 1 (top ranked) is matched again 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 tuple two at a time, and produce an appropriate `either` kind for the winner of that matchup. These kinds are combined by independent mixture into a new tuple. The first round has dimension 8; the second will have dimension 4; the third dimension 2; and the finals dimension 1.

```
@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):
        hi, lo = players[i], players[i + 1]
        weight = as_quantity('1.15') ** (lo - hi)
        k = k * either(hi, lo, weight)
    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
finals = next_round // third_round
```

Look at these in the playground; the act as we expect. For example, the combination $\langle 1, 2, 3, 4\rangle$ is most likely to come out of the first round, with $\langle 1, 2, 3, 5\rangle$ 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 match-ups 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
```

Now, we can compute our expectations:

```
E_finals = E(finals)
E_bottom_half = E(finals ^ (__ > 4))
```

which give values of about 3.17 and just under 0.25, respectively.

The next example illustrates a concept that arises frequently when we think about random processes: the concept of *state*. How do we describe the configuration, or *state*, of a system at a given moment? We need enough information in the state to understand the future evolution of the system from that 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, but 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.

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 analyze 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. The conditioning operation uses the weights in `state` to average the corresponding kinds in `move` to compute the (unconditional) *kind* of the next state. The result `escaped`, which is approximately 0.368, is the probability that the mouse escapes before the cat catches it.

For the second approach, we use an expanded definition of state. 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)
```

```
@conditional_kind
def move_alt(attempts_and_step):
    n_attempts, step = attempts_and_step
    if step == 3 or step == -1:
        return constant(attempts_and_step)

    n = n_attempts + 1
    if n_attempts == 16:
        if step < 2:
            return constant(n_attempts, -1)
        else:
            return either((n, -1), (n, 3), 2)

    if step == 0:
        return either((n, 0), (n, 1), 2)

    return either((n, step - 1), (n, step + 1), 2)
```

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.

```
state = initial_state_alt
for _ in range(16):
    state = move_alt // state
escaped_alt = E(state ^ (Proj[2] == 3))
```

which gives us the same answer.

9.4 Choices and Representations

Despite their power, neither computation nor mathematics are fully "automatic." We often need to apply creativity to get useful results even when the models themselves have simple descriptions. In this subsection, we look at examples where we use a little flare to select the choices we make or the representations of the data we use in answering a question.

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 choice of strategies for making the decision.

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 in which you are free. Assume that all ordering of the appointments have the same weight, i.e., are equally likely.

You have a very particular set of criteria in mind for the position. At each interview, you speak to the applicant and evaluate their qualifications with respect to these criteria. Consequently, after each interview, you can unambiguously rank the applicants you've seen thus far as to their suitability for the job. However, this is a demanding job market, 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.

How well can you do? What selection strategy should you use? For this example, consider the "After k" strategy 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. It is possible under this strategy to hire no one.

Notice that there are n different strategies here: "After 0", "After 1", "After 2", ..., "After $n-1$". In the "After 0" strategy you would always choose the first applicant.

Find the probability of selecting the *best* applicant out of the n using the "After k" strategy for each $k \in [0\,..\,n-1]$. Which of these strategies is most likely to select the best assistant?

First, we can reason that because all orderings of the applicants have the same weight, the position of the best candidate among n has a kind with equal weight on all n possible positions. We create a conditional kind that returns the kind of the position of the best candidate *before* the given value.

```
best_position_before = conditional_kind(
    lambda m: uniform(irange(1, m - 1)) if m > 1 else constant(0),
    codim=1)

best_position = conditional_kind(lambda n: uniform(irange(1, n)), codim=1)
```

For instance, the best position out of n is the value of an FRP with kind `best_position_before(n + 1)`. We have the boundary condition that returns a constant kind at 0 if there is *no* position before, such as when $n + 1$ is 1.

Whether the "After k" strategy succeeds (i.e., finds the best candidate) really depends on two pieces of information: the position b of the best candidate and the position s of the second-best candidate *up to and including position* b. The strategy succeeds if the best candidate is after k but the "second-best" candidate's position is not after k. This looks like:

```
def after_k(k: int):
    @statistic
    def best_before_k(b, s):
        return b > k and s <= k
    return best_before_k
```

The function `after_k` gives a statistic for each k.

To get the positions of the best candidate (b) and the second-best candidate up to b, which we'll call s, we apply the same reasoning twice. In particular, the best candidate before b is equally likely to be in any of those $b - 1$ positions. Hence,

```
best_position_before(n + 1) >> best_position_before
```

is the kind with values $\langle b, s \rangle$.

We can thus calculate the probability of selecting the best candidate for each "After k" strategy:

```
def assistant(n):
    "Solves the assistant problem with `n` candidates."
    probs = []
    for k in irange(0, n - 1):
        found_best = after_k(k)
        success = found_best(best_position_before(n + 1) >> best_position_before)
        probs.append(as_scalar( E(success) ))
    return index_of(max(probs), probs), probs
```

We find the probability of finding the best candidate for each strategy. The application of `as_scalar` to the expectation unwraps the number from the tuple. We gather these probabilities and find which k gives the maximum.

If we look at the best k for each n, we see that $k \approx ne^{-1}$. We will see later where this connection 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.

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 has value representing the number of passengers entering the elevator; its kind is given below.
- Each passenger's chosen floor is an FRP with kind `uniform(1, 2, ..., n)`.
- All passenger choices are independent of each other.

If no passengers enter, then the elevator makes zero stops.

The FRP S has as value the number of stops the elevator makes. Find $\mathbb{E}(S)$, assuming $\mu = 5$ and $n = 10$. The kind of N is given by

```
max_floor = 10
mu = 5
```

```
customer_weights = accumulate([mu / j for j in range(1, 21 + 1)],
                              func=mul, initial=1)
kind_N = weighted_as(0, 1, ..., 21, weights=customer_weights)
```

The accumulate here (from the standard python `itertools` package), builds a sequence of successively multiplied values, giving $\mu^j/j!$ in the jth position.

With 20 or more passengers possible, the kinds we need will be slow to compute because there are 10 possible floors for each passenger. Keeping track of which floor each passenger visits is not really necessary to answer the question we are asking. 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. We encode this set as a 10-tuple of 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. You should look at some of these kinds in the playground to get a feel for the representation.

Each passenger chooses among the available floors (from 1 to the top floor of the building) with equal weight:

```
passenger_floor = uniform(1, 2, ..., n)
```

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 form to another. We implement these as *functions* 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

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.

```
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])
floors = 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.

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*. 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_visited_floors = Sum(floors) // kind_N
E(number_visited_floors)
```

The kind `number_visited_floors` looks like

```
    ,---- 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
```

and has expectation ≈ 3.935.

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.

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.

Find the expectation of the total damage during the year given that there were n severe waves.

Let T be the FRP whose value is the total damage that accumulates in a year. Find $\mathbb{E}(T)$.

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 viewed conditional on 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,

$$\mathsf{kind}(T \mid N = n) = \mathsf{Sum}(\mathsf{kind}(D_1) \star\star\, n).$$

On the left, we have a conditional kind (given the value n of N) and the right an exact expression for it. We can express the right hand side as:

$$\mathsf{Sum}(\mathsf{kind}(D_1) \star\star\, n) = \mathsf{kind}(D_1 + D_2 + \cdots + D_n),$$

and because of the additivity property of expectation, we have that

$$\mathbb{E}(T \mid N = n) = \mathbb{E}(D_1 + D_2 + \cdots + D_n) = n\mathbb{E}(D_1),$$

because all the D_i's have the same kind.

We now have two ways to solve this. The simplest way applies conditioning to the $\mathbb{E}(T \mid N = n)$, which we can view as a conditional kind giving a constant for every n. That is:

$$\mathbb{E}(T) = \sum_n p_n \, \mathbb{E}(T \mid N = n),$$

where p_n is the weight on n in $\mathsf{kind}(N)$.

The second equivalent way is to apply conditioning to $\mathsf{kind}(T \mid N = n)$, conditioning on N, and then take expectations. This gives the same answer.

Here are the two methods implemented in the playground.

```
def rig_at_risk_a(kind_N, kind_D):
    """Computes expectation of total wave damage by mixing conditional *expectations*.
    Parameters `kind_N` and `kind_D` give arbitrary kinds for the number of waves
    and the damage per wave.

    """
    @conditional_kind(codim=1)
    def E_damage_given_n(n):
        return constant(n * E(kind_D))

    return E(E_damage_given_n // kind_N)

def rig_at_risk_b(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_alt = E(damage_given_n // kind_N)

rig_at_risk_a(uniform(1, 2, ..., 10), uniform(1, 2, 3))
```

```
rig_at_risk_b(uniform(1, 2, ..., 10), uniform(1, 2, 3))
```

Both give an expectation of 11 for these particular input kinds.

9.5 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.

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.

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.

```
p = symbol('p')
girl = either(0, 1)
at_least_one_girl = Or(Proj[1] == 1, Proj[2] == 1)
both_girls = And(Proj[1] == 1, Proj[2] == 1)

outcome_no_names = girl * girl | at_least_one_girl
```

The kind of the raw outcome is `girl * girl`, which looks like

```
    ,---- 1/4 ---- <0, 0>
    |---- 1/4 ---- <0, 1>
<> -|
    |---- 1/4 ---- <1, 0>
    `---- 1/4 ---- <1, 1>
```

which we then constrain with the conditional `at_least_one_girl` to produce the kind

```
    ,---- 1/3 ---- <0, 1>
<> -+---- 1/3 ---- <1, 0>
    `---- 1/3 ---- <1, 1>
```

The conditional eliminated the `<0, 0>` branch. Note that the relative size of the weights in the remaining branches *does not change*. When we re-normalize three equally weighted branches, we get the weights shown. The information that there is at least one girl has simply ruled out the possibility that both of the children are boys. And hence:

```
both_girls(outcome_no_names)
    ,---- 2/3 ---- 0
<> -|
    `---- 1/3 ---- 1
```

with `E( both_girls(outcome_no_names) ) = 1/3`.

The information we have in the second part is somewhat different; we know that at least one of the children is a girl with a rare name. So we have:

```
rare = weighted_as(10, 11, weights=[1 - p, p])
neighbors = girl * girl * rare * rare
a_rare_girl = Or(And(Proj[1] > 0, Proj[3] > 10), And(Proj[2] > 0, Proj[4] > 10))
outcome = neighbors | a_rare_girld
```

which looks like

```
    ,---- (-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>
```

Taking expectations `E(both_girls(outcome))`, we get $\frac{2-p}{4-p} \approx \frac{1}{2}$, since p is small.

This is a surprising difference, but we can get insight by studying the above kind. Since p is small, we can take it, for this purpose, to be so small that we can ignore it:

```
  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>
```

The resulting 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.*

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 for not-green and green.

```
bucket = either(0, 1)
green_given_bucket = conditional_kind({
    0: either(0, 1, 9),
    1: either(0, 1, 4)
})

which_bucket = Proj[1]( bucket >> green_given_bucket | (Proj[2] == 1) )
```

The kind `which_bucket` is the kind of the FRP that indicates which bucket (0 for left, 1 for right) given that we have observed a green ball. 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 get `E(which_bucket) = 5/8`.

The previous example is a demonstration of a common and important pattern call, **Bayes's Rule**. 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 corresponds 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 $\mathsf{kind}(X)$, and for any possible value u of X; we know $\mathsf{kind}(Y \mid X = u)$; and we have observed a value v of Y. Note that the mapping from u to $\mathsf{kind}(Y \mid X = u)$ is a *conditional kind* that we can denote by $\mathsf{kind}(Y \mid X)$.

Bayes's Rule is equivalent to three steps:

1. Build with a mixture the kind of the combined outcome $\langle X, Y\rangle$:

$$\mathsf{kind}(\langle X, Y\rangle) = \mathsf{kind}(X) \triangleright \mathsf{kind}(Y \mid X).$$

2. Constrain this with a conditional using the observation that $Y = v$:

$$\mathsf{kind}(\langle X, Y\rangle) \mid Y = v$$

3. Transform with a projection statistic to extract the X components $1, \ldots, \mathsf{dimension}(X)$:

$$\mathsf{proj}_{1..\,\mathsf{dimension}(X)}\left(\mathsf{kind}(\langle X, Y\rangle) \mid Y = v\right).$$

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 $\mathsf{kind}(X)$ and $\mathsf{kind}(Y \mid X)$, we have

```
def bayes(observed_y, x, y_given_x):
    i = dim(x) + 1
    return (x >> y_given_x | (Proj[i:] == observed_y)) ^ Proj[1:i]
```

So for example, in the previous example, we could write

```
which_bucket = bayes(1, bucket, green_given_bucket)
```

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.

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 $\mathsf{kind}(D)$ and $\mathsf{kind}(T \mid D)$. Notice that this kind and conditional kind are natural to specify from the assumptions descried.

```
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:

```
E(bayes(1, has_disease, test_by_status))
```

The result is

$$\frac{pd}{pd + (1-n)(1-d)}.$$

We can understand this by looking at the combined kind of $\langle D, T\rangle$:

```
    ,---- 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 we found 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(bayes(1, has_disease, test_by_status)),
             d='1/1000', n='99/100', p='95/100')
# is 0.0868
substitution(E(bayes(1, has_disease, test_by_status)),
             d='1/10_000', n='99/100', p='95/100')
# is 0.0094
substitution(E(bayes(1, has_disease, test_by_status)),
             d='1/10_000', n='999/1000', p='999/1000')
```

```
# is 0.0908
substitution(E(bayes(1, has_disease, test_by_status)),
             d='1/10_000', n='95/100', p='9/10')
# is 0.0018
substitution(E(bayes(1, has_disease, test_by_status)),
             d='1/100', n='95/100', p='9/100')
# is 0.1538
substitution(E(bayes(1, has_disease, test_by_status)),
             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: 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.

9.6 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.

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 whose value is 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, $\mathsf{kind}(H)$?

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.

At any point, we can be in one of two states: we’ve seen a heads or we haven’t. So on any flip, if we get a heads, we have made one flip and are done. If we get a tails, then we have made one flip and are in the same state before the flip, with the *kind* of our *remaining* number of flips is the same as $\mathsf{kind}(H)$.

We can capture this idea with a conditional kind. Suppose `remaining_flips` is the *kind* of the number *additional* of flips we will have to wait for a heads *after* the current flip. Then the kind of the total number of flips we have to wait *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 plus the remaining flips if we get a tails on 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
```

With `wait_for_heads`, we can approximate the kind of the number of flips and use it to get exact answers to our questions. As we will see, we can even use it to get an exact answer kind as well, even though that kind will have an infinite number of possible values.

If we put a reasonable *guess* as to the kind of the remaining flips, `wait_for_heads` will bring us closer. Look at the kind `guess` in the playground before each call to `wait_for_heads` in the following:

```
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.

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))`.

Using this, we can compute exact probabilities of waiting *more than n flips* and of waiting *exactly n flips*:

```
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 nth flip, we need to get $n-1$ tails followed by a heads.

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. This makes sense.

We can find this exactly with a bit of reasoning. In particular, going back to the `wait_for_heads` function, the argument `remaining_flips` is kind of the number of flips we have to wait after a tails. We put in a guess for that earlier, but we know that that kind is exactly the kind we want to find. In other words:

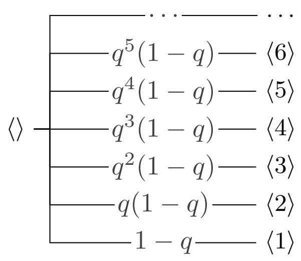
```
remaining_flips == wait_for_heads(remaining_flips)
```

The kind we seek should not be changed by applying `wait_for_heads`. Hence, the two sides of this equation give us *the same kind*. 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 24 and 25 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 kind (ha ha) of analysis with slightly different situations.

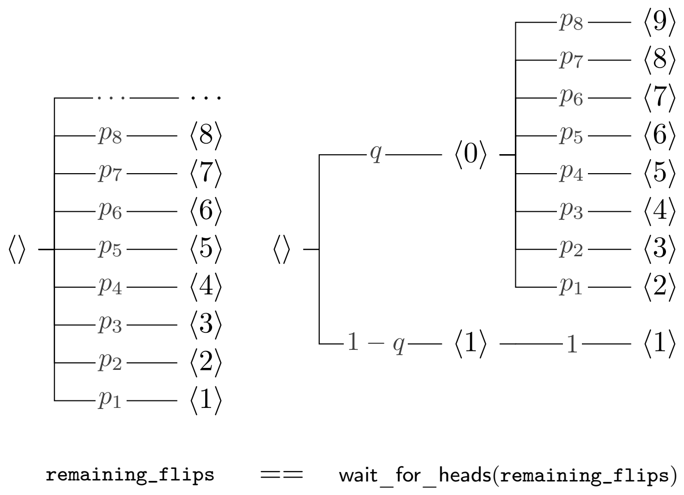


FIGURE 24. Solving for `remaining_flips`. On the left is the kind `remaining_flips` with arbitrary weights assigned to each value. We want to solve for those weights in terms of q. On the right is the value of `wait_for_heads(remaining_flips)` which operates by conditioning on a single flip. (See that function earlier.)

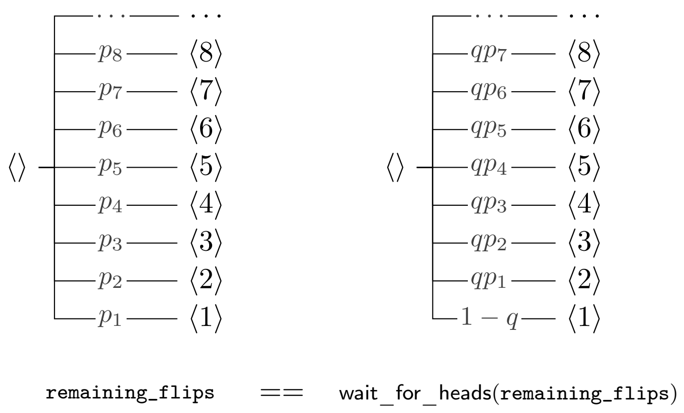


FIGURE 25. Solving for `remaining_flips` continued. Here, we reduce the right-hand kind in Figure 25 to canonical form. Making the two kinds here equal just requires equating the waits on the branches for each value. So, $p_1 = 1-q$, $p_2 = qp_1$, $p_3 = qp_2$, and so on. We can solve these equations for the p_i's as shown in the text.

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 different possibilities. 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), 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

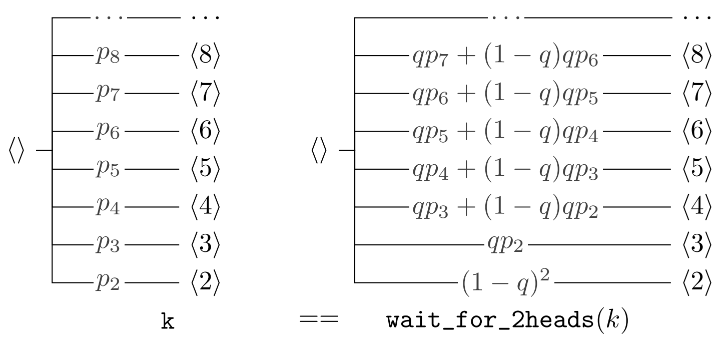
```
def wait_for_2heads(remaining_flips, q=symbol('q')):
    """Returns the kind of the total number of flips to get consecutive heads.

    `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.



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:

$$\begin{aligned}
p_2 &= (1-q)^2 \\
p_3 &= q(1-q)^2 \\
p_4 &= q^2(1-q)^2 + q(1-q)^3 \\
&\vdots
\end{aligned}$$

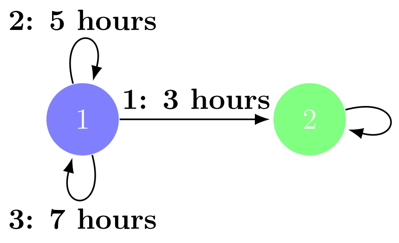
for all integers $j \geq 2$.

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.

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 Finite-State Machine with two states the outgoing transitions from a state correspond to channels.



Let T and C be FRPs. We interpret T's value as the # of hours until the robot reaches base, and C's value as the channel (1, 2, or 3) that the robot chooses *initially*. We want to find $\mathbb{E}(T)$.

```
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:

```
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. We see that quite quickly, the expectation converge to 15.

Can we solve this exactly? Yes, but the same as idea as before, but even simpler if we only need the expectation.

In particular, the kind of the robot's time to base is the (infinite) kind `t` that satisfies the equation:

```
  t == time_to_base(t)
```

which necessarily means that `E(t) == E(time_to_base(t))`.

But `time_to_base`, just does a conditioning operation, conditioning on a `uniform(1, 2, 3)` kind. Hence,

$$\begin{aligned}
\mathbb{E}(\texttt{time\_to\_base}(t)) &= \frac{1}{3}\,\mathbb{E}(\texttt{constant}(3)) + \frac{1}{3}(\mathbb{E}(t) + 5) + \frac{1}{3}(\mathbb{E}(t) + 7) \\
&= 1 + \frac{2}{3}\,\mathbb{E}(t) + 4 \\
&= 5 + \frac{2}{3}\,\mathbb{E}(t).
\end{aligned}$$

Solving our equation $\mathbb{E}(t) = 5 + \frac{2}{3}\,\mathbb{E}(t)$ gives us $\mathbb{E}(t) = 15$ as expected.

Note that `E(t ^ (__ + 5)) = E(t) + 5` by the Additivity property in equation (8.4) and similarly with 7.

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, either alone or in combination to compute predictions.

10 Building Models

Key Take Aways

FRPs are wonderful models of *Finite Random Processes*: random systems that evolve in finite time with a finite number of possible configurations or outcomes. In fact, as we have seen, we can push FRPs beyond the finite some careful reasoning/computation.

Restricting ourselves to finite systems, however large, has limitations in practice. So we will extend our models to more general random processes, with FRPs as a special case.

The key point here is that **all the operations and techniques we have used carry over to the more general case**. Handling infinities does raise a few technical challenges that impact how we handle some calculations (e.g., sums versus integrals), but the structure of the operations does not change.

We build models for random systems by using FRPs to describe observations we make of that system as it evolves. We assign meaning to the numeric values produced by the FRPs that connects them to the system we are observing. Our analysis will reduce to finding the kind or expectation of some compatible statistic. We sometimes want to make decisions or perform interventions or set parameters *as the random process evolves*, and we do that by having an FRP and compatible statistic for each possible decision or intervention or parameter setting. Our analysis guides us to choose a decision/intervention/parameter that optimizes some goal

A model is in effect a collection of assumptions about the random process that generates our observed data, and our analysis – and its guidance – are predicated on those assumptions. For this reason, we often consider a range of models with assumptions of varying strength. But in the end, we balance the reasonableness of the assumptions with the strength of the conclusions that we can draw from them. *Without assumptions, there can be no conclusions.*

Here, we also take a first look at infinite kinds and see how the different operations on them end up being just what is familiar from what we have discussed so far.

We have developed a rich set of tools for calculating with FRPs and kinds, and we have applied them to analyze a wide variety of random systems. Our goal in this section is to look more closely at the modeling process and to extend it to capture even more general systems.

FRPs are ideally suited as models of *Finite Random Processes*, the evolution of systems over *finite* horizons of time and space, with a *finite* number of possible configurations or outcomes, and with *finite* dimension. In reality, everything we work with is finite in this way, but in practice, it is sometimes easier – mathematically or computationally or both – to frame our models that allow infinities in horizon, size, or dimension. So, we will extend our tools to allow for more general random processes, which will include *infinite kinds*. Critically we will see that ***all the operations we have been using, the Big 3+1 and operations derived from them, carry over directly to the general case***. In our calculations, we will have to handle a few practical and technical details involved with managing infinities, but the conceptual structure of our analysis will not change.

We start this section by considering our modeling process in the abstract. This is the process we have used in all the examples so far, and it is the one we will continue to use for more general processes. Then we consider the practical limitations of finite models and begin to develop a representation of infinite models.

10.1 Building a Model

An FRP produces one value, one time. In modeling a random system, we construct an FRP whose value represents the *full set of data that we will measure* from the system as it evolves. Our analysis then proceeds by focusing on a statistic that operates on these data and returns a value that helps us answer our questions.

In the Doubled Cards example on page 124, for instance, the full data is the FRP $\langle F, S\rangle$ giving the values of the first and second cards drawn from the deck. The statistic of interest, `is_card_doubled`, extracts from those data the answer to our key question in the form of a transformed FRP $\texttt{is\_card\_doubled}(F, S)$.

In the Hunter's Success example on page 126, the full data is the FRP $\langle H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8\rangle$, and we get our answer by applying the Sum statistic, to get the transformed FRP H.

Sometimes, however, we want to make decisions or perform interventions or set parameters *as the random process evolves*, and we want our model and analysis to guide us to good choices.

In the Monty Hall example of Section 2.3, for instance, you select a *strategy* that helps determine the outcome of the game. The full data is the FRP $\langle M, Y \rangle$ giving the door with the prize and the door you select. The game outcome extracted from the data by a statistic that depends on the strategy (for instance, on whether you switch).

Here is a schematic for our modeling process that handles the full range of cases:

The word "decisions" here is meant in a broad sense and can encompass choices, strategies, settings, configuration, and actions that we specify at various points as the process evolves. The key is that the process's evolution is influenced by this choice.

1. We describe the decisions that we have by a space Decisions. Each $\theta \in$ Decisions fully describes the choices available to us.

2. To each $\theta \in$ Decisions, we associate an FRP X_θ whose value represents the full data we will observe or measure from the process.

3. We relate the data to the questions we want to answer with a statistic ψ_θ that is compatible with X_θ. The value of the transformed FRP $\psi_\theta(X_\theta)$ answers those questions.

4. Using our knowledge and assumptions about the process under study, we make predictions about $\psi_\theta(X_\theta)$ by calculating either $\mathsf{kind}\,(\psi_\theta(X_\theta))$ or $\mathbb{E}\,(\psi_\theta(X_\theta))$.

5. We choose a value of θ that gives us the best predicted outcome.

6. We then observe the process making decisions based on our chosen θ and observe the value of X_θ.

In cases like Doubled Cards, there are no decisions to account for, so we can take Decisions $= \{1\}$, in which case we have just one FRP and one statistic. In cases like Monty Hall, each element θ of Decisions is a pair: the door you choose and whether you switch.

Texas hold 'em In the poker variant *Texas hold 'em*, each player receives two cards, dealt face down (the first stage of the game). All players share five cards that are dealt face up in stages (three cards in the second stage, one card in the third stage, and one card in the fourth stage). A round of betting takes place after each stage.

1. The decision space for this system includes all the betting choices (amount and type of bet, e.g., call, raise, fold) available to each player at each stage of the game. Call this a *strategy* for the game.

2. The full data are represented by an FRP whose value represents all players hands with the cards in order that they receive, and the shared cards, also in order

3. The statistic we use depends on the question we are asking. If, for example, you are one of the players, you might want to use a statistic that indicates whether you win the hand.

4. We compute the expectation of our statistic, such as the probability that you win the hand, as a function of the strategy.

5. We choose the strategy to optimize the outcome, such as maximizing your probability of winning.

6. We play the hand with the chosen strategy and observe the outcome.

We will use the same modeling process throughout. The decision space, full data structure, and statistics will change to suit the situation, but the shape of the analysis and the techniques we use will not.

10.2 Limitations of Finite Models

Finite Processes are good models for many situations, but they have some practical limitations even in some simple cases:

- Large finite sets of quantities can be harder to work with in practice than infinite sets.

- We are often interested in processes that run over arbitrarily large spans of time or space.

- Exact answers are often easier to obtain when we allow infinite sizes.

- Computation can be more difficult with finite configurations than with some kinds of infinite configurations.

This may seem counter-intuitive; after all, infinity is rather big. In principle, we can handle any finite system, but in practice, computational limitations get in the way.

Consider a simple example: taking a weighted average over all permutations of a deck of cards. 52! is a big number, and whether on a computer or in a mathematical expression, we will only be able to compute the value of that average if we can rely on some reasoning to simply the result into a more manageable form.

Similarly, if we try to compute the kind of an FRP whose values could be any number between 0 and 1 up to eight decimal places, we have a size bigger than 100 million. To compute the expectation of that kind requires those terms and even storing it will require substantial space. This is feasible to a point but increasingly inconvenient. What option do we have? In practice, we can represent such large kinds by *functions on real numbers* where the large sum becomes an integral. And in many cases, as we will see, that is both easier to represent and to compute.

And as a final example, consider the Waiting for Heads example on page 153. We can artificially restrict the number of flips to consider, say to 100. For most values of p, the probability of heads on a flip, it is very unlikely that it will take more than 100 flips to see a heads. So a model with a limited number of flips is a good approximation, but such a model forces us to choose the number of flips according to the chance of heads. It also gives us more complicated answers. The kind we found in the example has infinite size, but the weights have a very simple form. So again, in an important sense, an infinite model is easier to handle.

It is important to emphasize that there are *two types of infinity* that we need to work with. The first is like the integers: each element a discrete item that we can count one by one, though the count goes on forever. Any infinite set of values such that every element can be associated with a distinct natural number has this type of infinity. We call such an infinite set **discrete**, or countable. The second is like the real numbers: the elements are like molecules of a fluid that slip between your fingers when you try to hold them, though we can hold them in a cup. The elements of an infinite set like this cannot be counted like the integers, even if we had the time; any list we made would be incomplete We call such an infinite set **continuous** or uncountable.

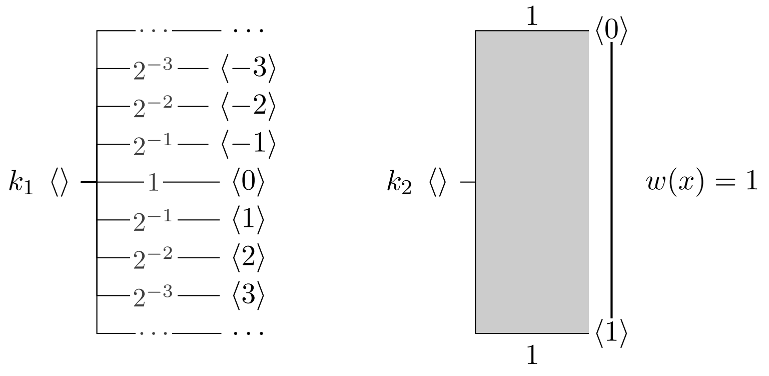
We will build models with both types of infinities, and fortunately, all the operations we have been using (**the big 3+1!**) will still apply.

10.3 Infinite Kinds

The kind of an FRP associates a positive weight to each possible value of an FRP. Kinds will have the same structure even when they have infinite size: we associate a positive weight to each value. There are only differences: i. with kinds of infinite size, we require that total weight is finite so that we can standardize the weights, and ii. our interpretation of the weight changes slightly when the kind is continuous, as we will see.

The figure below shows two kinds with infinite size, the one on the left discrete, the one on the right continuous. The discrete tree has a branch for every integer and positive weights. The one constraint on the weights is that they have a finite sum, which means that the weights must get smaller as we move out to more distant branches. And these do, summing to 3 over all the integers. Beyond this constraint, there is little difference how we work with the kind.

The continuous tree on the right looks different because there is a branch (and corresponding value) for every real number between 0 and 1. There are so many branches that we cannot distinguish them, so they appear as a gray rectangle. The line from the value ⟨0⟩ to the value ⟨1⟩ represents the continuum of values.



The weights on the continuous kind are all 1, but their interpretation is slightly different than what we are used to. A good analogy is that the weights on a finite or infinite discrete are in units of *mass* whereas the weights on a continuous kind are in units of *density*. We will explore the implications of that difference soon.

10.4 Kernels and Kinds

We represent kinds as a one-level tree with distinct leaves. It is useful to have more than a pictorial representation, and what we use is a *function*.

The **kernel** of a kind, finite or infinite, is a function K that associates values with weights. Specifically if X is an FRP or a more general random process, then K_X, the kernel of X's kind (or just the kernel of X for short) associates weight $\mathsf{K}_X(v)$ to value v.

It will be useful to have a name that encompasses an FRP and the analogue for a more general random process. Both are devices that produce one value, one time when activated, though in the more general case, the value may not be infinite dimensional. We will call such a device, either an FRP or a more general version, a **random variable**.

For any random variable X, its kernel K_X defines a kind. Any value v for which $\mathsf{K}_X(v) > 0$ is called a **possible value** of X. It is often convenient to allow K_X to be defined for other values as well; a value v with $\mathsf{K}_X(v) = 0$ is a value with no branch in the tree.

Our next task is to see that the Big 3+1 can be expressed in terms of the kernels, whether finite or infinite, and that these operations look and act the same way as we are used to.

Checkpoints

After reading this section you should be able to:

- Describe the process by which we build a model for a random system, including models that allow us to make decisions as they evolve.
- Explain the challenges that can arise in dealing with large finite kinds.
- Represent a kind, finite or infinite, as either a tree or a kernel function.
- Define a random variable, a kernel, and a possible value.