frplib Cookbook

Contents

v1.0.0 Last Updated 08 Sep 2024

This Cookbook illustrates how to perform various common tasks in `frplib`. These can be used either in the frp playground or in Python code that imports frplib. In the latter case, you need to import any frplib symbols you use, see Recipe 26, "Importing `frplib` Objects in Your Code." To show the results of some commands, we will precede the commands with the `pgd>` prompt and display the result after the command.

Recipe 1. Constructing Kinds with Kind Factories

Although FRPs are fundamental in the sense that they represent the actual the random outputs of a system, in practice, we often work with Kinds, both to construct FRPs and for many calculations. So we start by showing how to use *Kind factories* to construct Kinds.

A Kind factory is just a function that takes as input some parameters and returns a Kind that is consistent with those parameters. The Kind factories discussed here include `constant`, `either`, `binary`, `uniform`, `weighted_as`, `weighted_pairs`, `weighted_by`, `evenly_spaced`, `linear`, `symmetric`, `geometric`, `without_replacement`, and `arbitrary`.

The simplest factory `constant` produces a Kind with only one possible value, which can be given as a single tuple or multiple arguments which become its components. We call it with

```
constant(value...)    # produces   <> ----- 1 ---- <value...>
```

For example:

```
pgd> constant(0)
<> ------ 1 ---- 0
pgd> constant(1, 2, 3)
<> ------ 1 ---- <1, 2, 3>
pgd> constant((10, 20))
<> ------ 1 ---- <10, 20>
```

We often use Kinds with two values. The `either` Kind factory takes two values and an optional ratio of weights (first value to second), which defaults to 1. The `binary` factory is a special case where the values are 0 and 1 and the weight on 1 is given as $0 < p < 1$.

```
either(a, b, r)        # r is the ratio of weights a to b
binary(p)              # v0.2.4+
```

For example:

```
pgd> either(1, 2)
    ,---- 1/2 ---- 1
<> -|
    `---- 1/2 ---- 2
<> ------ 1 ---- 0
pgd> either(1, 2, 9)
    ,---- 0.9 ----- 1
<> -|
    `---- 0.1 ----- 2
pgd> either(1, 2, '1/9')
    ,---- 0.1 ----- 1
<> -|
    `---- 0.9 ----- 2
pgd> binary()  # p = 1/2 by default, either(0, 1)
    ,---- 1/2 ---- 0
<> -|
    `---- 1/2 ---- 1
pgd> binary('3/4')
    ,---- 1/4 ---- 0
<> -|
    `---- 3/4 ---- 1
pgd> binary('1/4')
    ,---- 3/4 ---- 0
<> -|
    `---- 1/4 ---- 1
```

A very common case is to construct a Kind with *equal weights* on all its values. This is produced by the `uniform` Kind factory. Like most of the Kind factories, `uniform` accepts either a single sequence of values (e.g., a Python list or set or other iterable) or values given as multiple arguments

```
uniform({1, 2, 3})
uniform([(0, 0), (0, 1), (1, 0), (1, 1)])
uniform(1, 2, 3)
uniform((0, 0), (0, 1), (1, 0), (1, 1))
uniform((x, y) for x in [0, 1] for y in [0, 1])
```

In the first two cases, we give the values as a single set (enclosed in {}s) or a single list (enclosed in []s). In the third and fourth cases, the values are given as separate arguments. In the last case, we pass a Python generator expression to construct the values dynamically rather than write them out explicitly.

Like most Kind factories, scalar values passed to `uniform` can include a `...` that extends the two values before the `...` forward up to but not including the value after the `...`. $a, b, \ldots, c$ gives $a, b, b + (b - a), b + 2(b - a), \ldots, c$. For example,

```
uniform(1, 2, ..., 6)
uniform(10, 20, ..., 100)
uniform(9, 8, ..., 1)
uniform(1, 4, 5, 9, 11, 13, ..., 22)
```

The latter has values 1, 4, 5, 9, 11, 13, 15, 17, 19, 21, and 22.

More generally, we want to specify values *and* weights. Three factories make this easy:

```
weighted_as(values..., weights=weight_list)
weighted_pairs([(value1, weight1), (value2, weight2), ...])
weighted_by(values..., weight_by=function)
```

The `weighted_as` factory is our workhorse; the weights are specified by a list or other sequence. `weighted_pairs` takes a Python list of pairs, containing values and their corresponding weights. And `weighted_by` takes the values and a function that assigns the weight to each value by calling `function(value)`, where value is passed as is, without any translation or conversion.

The values and weights in these factories can be any *quantity*, see Recipe 28, and the weights can use `...` patterns as described above.

For example:

```
weighted_as(1, 2, ..., 6, weights=[10, 11, ..., 15])
weighted_pairs([((1, 1), 3), ((1, 0), 2), ((0, 1), 2), ((0, 0), 1)])
weighted_by(1, 2, ..., 100, weight_by=lambda v: v * v)
```

Related is the Kind factory `evenly_spaced`, which specifies evenly-spaced scalar values weighted with a `weight_by` function like `weighted_by`

```
evenly_spaced(0.2, 1.0, num=5, weight_by=lambda v: 1 + 5*v)
evenly_spaced(4, num=5)
```

With only one value given, it starts at 0, so `evenly_spaced(4, num=5)` is equivalent to `uniform(0, 1, ..., 4)`.

There are several other specialized factories for producing Kinds on the given values with specified patterns of weights:

```
linear(1, 2, ..., 6, increment=2)   # weights linear  first + increment * index
geometric(1, 2, ..., 6, r=0.5)      # weights geometric first r^index
symmetric(1, 2, ..., 6, around=3.5) # weights symmetric around aroud
```

Kinds can also be specified with *symbolic* values and weights. See Recipe 27.

```
pgd> a, b = symbols('a b')
pgd> either(a, b)
    ,---- 1/2 ---- a
<> -|
    `---- 1/2 ---- b
pgd> uniform(a, a + 1, a + 2)
    ,---- 1/3 ---- 1 + a
<> -+---- 1/3 ---- 2 + a
    `---- 1/3 ---- a
pgd> weighted_as(1, 2, weights=[a, b])
    ,---- a/(a + b) ---- 1
```

```
<> -|
    `---- b/(a + b) ---- 2
pgd> arbitrary(1, 2, 3)
```

The last of these produces a Kind on the specified values with arbitrary symbolic weights that can be named.

Finally, there are Kind factories that produce Kinds over special collections. For example, `without_replacement` gives the Kind of all subsets of a specified set from a collection of values.

```
pgd> without_replacement(2, [1, 2, 3, 4])
    ,---- 1/6 ---- <1, 2>
    |---- 1/6 ---- <1, 3>
    |---- 1/6 ---- <1, 4>
<> -|
    |---- 1/6 ---- <2, 3>
    |---- 1/6 ---- <2, 4>
    `---- 1/6 ---- <3, 4>
```

Recipe 2. Constructing FRPs with FRP Factories

A frequently used method for constructing FRPs is to specify its Kind to the `frp` function. This takes a Kind and returns a fresh FRP with that Kind.

```
X = frp(K)    # K is a Kind, kind(X) is K
```

Examples:

```
frp(either(1, 2))
frp(uniform(1, 2, ..., 1000))
frp(weighted_as((0, 0), (0, 1), (1, 0), (1, 1),
                 weights=['1/8', '1/8', '1/4', '1/2']))
```

The special FRP factory `shuffle` returns an FRP representing a random permutation of a given collection of values. For example, the following give an FRP representing a shuffle of a standard deck of cards.

```
shuffle(k for k in irange(1, 52))
shuffle(1, 2, ..., 52)            # v0.2.4+
```

Recipe 3. Cloning an FRP

If `X` is an FRP and

```
Y = clone(X)
```

then `Y` is a fresh FRP with the same Kind as `X`.

If `S` is a Conditional FRP and

```
T = clone(S)
```

then `T` is a Conditional FRP with the same inputs whose targets are clones of `S`'s targets.

Recipe 4. Checking if Two Kinds are Equal

Use `Kind.equal`:

```
Kind.equal(kind1, kind2)
```

returns `True` or `False` if the Kinds are equal (within numerical precision). This also works with Kinds whose values or weights are symbolic.

This takes an optional argument `tolerance` that specifies how close two numbers need to be to be considered numerically equal

```
Kind.equal(kind1, kind2, tolerance=1e-7)
```

Recipe 5. Constructing Kinds from FRPs

The function `kind` is used to find the Kind of an FRP or the Conditional Kind of a Conditional FRP.

```
kind(X)      # returns the Kind of FRP X
```

Note that some FRPs have Kinds that are computationally hard to compute. You can call `kind` on these FRPs but the computation will take a very long time. Given an FRP `X`,

```
X.is_kinded()
```

will give a Boolean indicating whether the Kind is available without additional computation.

```
pgd> frp(binary()).is_kinded()
True
pgd> Bits = frp(binary()) ** 100
pgd> Bits.is_kinded()
False
```

Recipe 6. Constructing Kinds from Strings

The `kind` function also accepts market-style strings. Do `info('kinds')` in the playground (or 'help kinds.' in the market) For example,

```
# equivalent to either(0, 1)
kind('(<> 1 <0> 1 <1>)'

# equivalent to weighted_as((0, 0), (0, 1), (1, 0), (1, 1),
#                              weights=[1, 1, 2, 4])
kind('(<> 1 <0, 0> 1 <0, 1> 2 <1, 0> 4 <1, 1>)'
```

Recipe 7. Finding Dimension, Codimension, Size, and Type

The functions `dim`, `codim`, `size`, and `typeof` extract information about Kinds, FRPs, Conditional Kinds, Conditional FRPs, and Statistics.

```
k = uniform(1, 2, ..., 100)
X = frp(k) ** 3
c = conditional_kind({0: constant(1, 2), 1: either((2, 3), (3, 4))})
C = conditional_frp(c)
s = Proj[1, 2, 3, 4]

size(k)         # Size of the Kind, 100

dim(k ** 2)     # Dimension 2
dim(X)          # Dimension 3
dim(s)          # Dimension 4

codim(X)        # Codimension 0 for any FRP or Kind
codim(c)        # Codimension 1
codim(C)        # Codimension 1
codim(s)        # Codimension 4 and up

typeof(k)       # 0 -> 1   dim -> codim
typeof(c)       # 1 -> 2
typeof(C)       # 1 -> 2
typeof(s)       # [4..) -> 1
```

Note that `size` applies only to Kinds and FRPs, and it forces the computation of an FRPs Kind.

Recipe 8. Running Demos of FRPs

The function `FRP.sample` provides the same functionality as the `demo` task in the frp market. It activates a large collection of fresh FRPs and tabulates their value.

```
FRP.sample(n, kind_or_frp, summary=True)
```

Accepts the size of the demo, a Kind or FRP, and an optional argument `summary` indicating whether to summarize the results (True) or list them individually (False).

```
FRP.sample(100, uniform(1, 2, 3))
FRP.sample(10_000, frp(geometric(0, 1, 2, ... 100, r=0.9)))
```

Recipe 9. Pruning Numerically Negligible Branches of a Kind

Some transformations with statistics can give branches with very small, numerically negligible weights, sometimes many of them. We use the `clean` to prune those branches and renormalize the Kind, which usually makes it nicer to view.

```
clean(k)
clean(k, tolerance=1e-7)
```

The optional argument `tolerance` gives the threshold for numerically negligible. By default it is small, 10^{-12}.

Recipe 10. Unfolding a Multi-dimensional Kind

A Kind with dimension > 1 can be unfolded to have width greater than 1 with the `unfold` function.

```
unfold(uniform(1, 2) * weighted_as(3, 4, weights=[9, 1]) * constant(5))
```

displays the unfolded tree Currently `unfold` does not work for Kinds with symbolic weights.

Recipe 11. Transforming FRPs and Kinds with Built-in Statistics

To transform a Kind `K` or FRP `X` with a compatible statistic `psi`, we can *either* apply the statistic or use the `^` operator:

```
psi(K)              psi(X)
K ^ psi             X ^ psi
```

Both forms are convenient in different situations and give the same result.

```
pgd> K = uniform(1, 2, 3) ** 2
pgd> X = frp(X)
pgd> X
An FRP with value <2, 3>
pgd> Sum(X)
An FRP with value <5>
pgd> K ^ (Proj[2] + 10)
    ,---- 1/3 ---- 11
<> -+---- 1/3 ---- 12
    `---- 1/3 ---- 13
pgd> X ^ Permute(2, 1)
An FRP with value <3, 2>
pgd> Max(K)
    ,---- 1/9 ---- 1
<> -+---- 3/9 ---- 2
    `---- 5/9 ---- 3
```

The `@` operator is related, see Recipe 22.

We can use the same operators to transform general conditional Kinds and FRPs. In this case, the statistic gets the input and target values and produces new target values.

```
pgd> Max(conditional_kind({1: constant(1), 2: either(0, 4)}))
A conditional Kind of type 1 -> 2 with wiring:
<1>:  <> ------ 1 ---- 1


          ,---- 1/2 ---- 2
<2>:  <> -|
          `---- 1/2 ---- 4
pgd> conditional_kind({1: constant(1), 2: either(0, 1)}) ^ (Proj[2] + 1)
A conditional Kind of type 1 -> 3 with wiring:
<1>:  <> ------ 1 ---- 2


          ,---- 1/2 ---- 1
```

```
<2>:  <> -|
          `---- 1/2 ---- 2
pgd> conditional_kind({1: constant(1), 2: either(0, 1)}) ^ (__ + 1)
A conditional Kind of type 1 -> 3 with wiring:
<1>:  <> ------ 1 ---- <2, 2>

          ,---- 1/2 ---- <3, 1>
<2>:  <> -|
          `---- 1/2 ---- <3, 2>
```

(See the `.transform_targets` method to give a statistic that just receives the target values.)

Recipe 12. Using Projections

Projections are statistics that extract one or more components from the tuple passed as input. The `Proj` factory constructs projection statistics, specified by indices or slices in []s.

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

Proj[3,5](10, 20, 30, 40, 50)    # == <30, 50>
Proj[1,3,5](10, 20, 30, 40, 50)  # == <10, 30, 50>

Proj[1:4](10, 20, 30, 40, 50)    # == <10, 30, 50> (item 4 excluded)
Proj[1:4:2](10, 20, 30, 40, 50)  # == <10, 50> (skip by 2)

Proj[3:](10, 20, 30, 40, 50)     # == <30, 40, 50> (to end)
Proj[:3](10, 20, 30, 40, 50)     # == <10, 20> (from start, 3 excluded)

Proj[-1](10, 20, 30, 40, 50)     # == <50> (-1 is last component)
Proj[-2:](10, 20, 30, 40, 50)    # == <40, 50>
Proj[:-2](10, 20, 30, 40, 50)    # == <10, 20, 30>
```

```
Proj[-1::-1](10, 20, 30, 40, 50) # == <50, 40, 30, 20, 10>
```

Recipe 13. Combining Built-in Statistics

Transforming a statistic with a statistic *composes* them.

```
pgd> X = uniform(1, 2, ..., 10) ** 3
An FRP with value <4, 9, 1>
pgd> X ^ Sqrt(Max)     # Computes Sqrt(Max(value))
An FRP with value <3>
pgd> X ^ Abs(Proj[1] - Proj[2])
An FRP with value <5>
```

Arithmetic operations on statistics produce statistics

```
pgd> X ^ (Proj[1] + 2 * Proj[2] + 3 * Proj[3])
An FRP with value <25>
pgd> X ^ (__ + 10)
An FRP with value <14, 19, 11>
pgd> X ^ Fork(Proj[2] - Proj[1], Proj[3] - Proj[2], Proj[1] - Proj[3])
An FRP with value <5, -8, 3>
```

Comparison operations on statistics produce conditions

```
pgd> X ^ (Proj[2] == 1)
An FRP with value <0>
pgd> X ^ (Proj[1] < 3)
An FRP with value <1>
pgd> X ^ (Proj[1] + Proj[2] > 12)
An FRP with value <1>
pgd> X ^ And(Proj[1] == 4, Proj[2] == 9, Proj[3] == 2)
An FRP with value <0>
```

Recipe 14. Defining Custom Statistics from Python Functions

We create statistics from Python functions by attaching a decorator – one of `@statistic`, `@scalar_statistic`, or `@condition` – before the definition. Each of these decorators accepts optional arguments (including `codim` and `dim`) that specify properties of the statistic.

```python
@statistic
def got7(rolls):
    return index_of(7, rolls)  # Index of first 7 or -1

@statistic(codim=1, dim=2)
def rotate90(x_y):
    x, y = x_y
    return (-y, x)

@scalar_statistic
def right_triangle_area(x, y):
    "area of right triangle with hypotenuse from origin to <x, y>"
    return (x * y) / 2
```

With two arguments, `frplib` infers that the codimension is 2, with one argument it allows any codimension unless specified. The `@scalar_statistic` decorator just ensures that the dimension is 1. The return value of a statistic is automatically converted to a vector tuple.

The `@condition` decorator converts a Boolean statistic into one that returns 0 (false) or 1 (true).

```python
@condition
def both_even(x, y):
    return x % 2 == 0 and y % 2 == 0
```

A condition always has dimension 1.

`statistic`, `scalar_statistic`, and `condition` can also be used as functions that take a function as an argument.

Chapter 0, Section 2 gives many more examples of custom statistics.

Recipe 15. Creating Independent Mixtures of FRPs and Kinds

The * operator gives independent mixtures of Kinds and FRPs (as well as more general conditional Kinds and conditional FRPs).

```
pgd> either(0, 1) * either(3, 4)
    ,---- 1/4 ---- <0, 3>
    |---- 1/4 ---- <0, 4>
<> -|
    |---- 1/4 ---- <1, 3>
    `---- 1/4 ---- <1, 4>
pgd> X = frp(either(0, 1))
pgd> X * clone(X)
An FRP with value <0, 1>
```

The `clone` is needed when using * on FRPs, but not with ** (see below).

An independent mixture of conditional Kinds and FRPs *with the same inputs* produces the type of object with the same inputs where the targets are the independent mixtures of the originals' targets. For exampe,

```
pgd> cK = conditional_kind({1: constant(1), 2: either(0, 4)})
pgd> cK * cK
A conditional Kind of type 1 -> 3 with wiring:
<1>:  <> ------ 1 ---- <1, 1>

          ,---- 1/4 ---- <0, 0>
          |---- 1/4 ---- <0, 4>
<2>:  <> -|
          |---- 1/4 ---- <4, 0>
          `---- 1/4 ---- <4, 4>
```

The ** operator computes *independent mixture powers*

```
pgd> uniform(2, 4, 6) ** 2
    ,---- 1/9 ---- <2, 2>
    |---- 1/9 ---- <2, 4>
```

```
     |---- 1/9 ---- <2, 6>
     |---- 1/9 ---- <4, 2>
<> -+---- 1/9 ---- <4, 4>
     |---- 1/9 ---- <4, 6>
     |---- 1/9 ---- <6, 2>
     |---- 1/9 ---- <6, 4>
     `---- 1/9 ---- <6, 6>
pgd> X ** 100
An FRP with value <1, 2, 3, 1, 3, 1, 2, 2, 3, 1, 1, 2, 1, 2, 3, 2, 1, 2, 1, 3, 2, 2, 2, 3, 1
1, 2, 2, 1, 3, 1, 1, 3, 3, 2, 3, 2, 2, 3, 1, 2, 1, 1, 2, 1, 2, 2, 2, 1, 2, 1, 2, 2, 2, 3, 2, 1
3, 3, 1, 1, 3, 3, 3, 1, 1, 1, 2, 3, 3, 3, 2, 3, 1, 1, 3, 3, 3, 3, 2, 1, 1, 2, 3, 3, 1, 1, 3, 1
evaluate its kind.)
```

For FRPs, `**` *automatically clones* the FRP.

Recipe 16. Computing Large Independent Mixture Powers of Kinds

Independent mixture powers of Kinds have sizes that grow exponentially in the exponent and so quickly become infeasible to compute in general. However, for some types of statistics, we can compute

```
psi(K ** n)
```

efficiently, where `psi` is a statistic, `K` is a Kind, and `n` is moderately large

These statistics are called "monoidal statistics" in `frplib` and include common statistics like `Sum`, `Count`, `Max`, and `Min`. The technique is discussed in detail in Chapter 0 Section 6.1. It is implemented in the `fast_mixture_pow`

```
fast_mixture_pow(stat, a_kind, n)  # computes stat(a_kind ** n)
```

For example,

```
fast_mixture_pow(Sum, either(0, 1), 100)
```

takes only a few seconds; done naively it would take longer than the age of the universe. (The `clean` operation is often useful after computing the Kinds of large mixture powers, see Recipe 9.)

Recipe 17. Defining Conditional FRPs and Kinds

We use `conditional_frp` to construct Conditional FRPs. This can be passed a dictionary

```
pgd> N = frp(either(0, 1, 99))
pgd> P = frp(either(0, 1, 1/19))
pgd> T = conditional_frp({ 0: N, 1: P })
```

or a function

```
def a_func(value):
    if value == 0:
        return N
    else:
        return P

# both of these give the same result
T = conditional_frp(a_func)
T = conditional_frp(lambda value: N if value == 0 else P)
```

or used as a *decorator* above a Python function definition.

```
@conditional_frp
def T(value):
    if value == 0:   # Inputs are scalars when codim=1
        return N
    return P
```

In all these cases, we get a conditional FRP object to which we can pass input, look at targets, and use for mixtures.

```
pgd> T.target(0)
An FRP with value <1>
pgd> T(0)
An FRP with value <0, 1>
pgd> frp(either(0, 1, 999)) >> T
```

```
An FRP with value <0, 1>
```

`conditional_frp` accepts optional parameters that specify the codimension, dimension, and domain of the conditional FRP. `frplib` can infer these from a dictionary and can infer the codimension from a function with multiple arguments, but otherwise it is good practice to supply at least the codimension if possible as it helps with error checking. The decorator can take these arguments as well.

```
@conditional_frp(codim=1, domain={0, 1})
def T(value):
    if value == 0:  # Inputs are scalars when codim=1
        return N
    return P
```

Analogously, we use `conditional_kind` to construct conditional Kinds with the same mechanisms: passing a dictionary

```
t = conditional_kind({
    0: either(0, 1, 99),
    1: either(0, 1, 1/19)
})
```

passing a function (named or anonymous)

```
t = conditional_kind(
        lambda value:
            either(0, 1, 99) if value == 0 else either(0, 1, 1/19),
        domain=[0, 1], target_dim=1
)
```

or putting a decorator on a function definition

```
@conditional_kind(domain=[0, 1], target_dim=1)
def t(value):
    if value == 0:
        return either(0, 1, 99)
```

```
    return either(0, 1, 1/19)
```

If `cF` is a conditional FRP, then

```
cK = conditional_kind(cF)
```

is the corresponding conditional Kind. This goes in reverse as well, with `conditional_frp` on `cK`, but a better way is to use the `kind` function which is a bit smarter about it

```
kind(cK)  # same as cF
```

Recipe 18. Building a Mixture of FRPs and Kinds

The mixture operator `>>` computes mixtures. The basic case is a mixture of a Kind `k` and conditional Kind `cK` `k >> cK` or of an FRP `X` and a conditional FRP `cF`, `X >> cF`. The dimension of the object on the left of the `>>` must be compatible with the codimension of the object on the right.

```
pgd> either(11, 22) >> conditional_kind({11: either(99, 100), 22: constant(7)})
    ,---- 1/4 ---- <11, 99>
<> -+---- 1/4 ---- <11, 100>
    `---- 2/4 ---- <22, 7>
pgd> X = frp(either(11, 22))
An FRP with value 22
pgd> cF = conditional_frp({11: frp(either(99, 100)), 22: frp(constant(7))})
A conditional FRP of type 1 -> 2 with wiring:
  <11>              An FRP with value 99
  <22>              An FRP with value 7
pgd> X >> cF
AN FRP with value <22, 7>
```

More generally, `>>` accepts a conditional FRP or Kind of type $m \to p$ and a conditional FRP or Kind of type $p \to n$ and produces a conditional FRP or Kind of type $m \to n$. Because an FRP or Kind of dimension m is just a conditional FRP or Kind of type $0 \to m$, this generalizes the basic case. For example:

```
pgd> cK1 = conditional_kind({0: either(1, 2), 1: either(3, 4)})
pgd> cK2 = conditional_kind({(0, 1): either(9, 10), (0, 2): either(11, 12),
                             (1, 3): either(21, 22), (1, 4): either(23, 24)})
pgd> cK1 >> cK2
A conditional Kind of type 1 -> 3 with wiring:
          ,---- 1/4 ---- <1, 9>
          |---- 1/4 ---- <1, 10>
<0>:  <> -|
          |---- 1/4 ---- <2, 11>
          `---- 1/4 ---- <2, 12>

          ,---- 1/4 ---- <3, 21>
          |---- 1/4 ---- <3, 22>
<1>:  <> -|
          |---- 1/4 ---- <4, 23>
          `---- 1/4 ---- <4, 24>
```

See Chapter 0, Section 4 for more on mixtures. And see Recipe 19 for the related *conditioning operator* `//`.

Recipe 19. Using the Conditioning Operator

The conditioning operator `//` is a combination of mixture and projection. We can think of it as an averaging operation where the target Kinds in a conditional Kind are averaged over the inputs according to the weights of a given Kind. We use it as

```
cK // K
```

where `cK` is a conditional Kind (or FRP) and `K` is a Kind (or FRP). This is equivalent to

```
K >> cK ^ Proj[(d+1):]
```

where `d == dim(K)`.

```
pgd> door = uniform(1, 2, 3)
pgd> prize_by_door = conditional_kind({
...>     1: either(-10, 100),
...>     2: either(-50, 250),
...>     3: constant(-100)
...> })
pgd> prize_by_door // door
     ,---- 2/6 ---- -100
     |---- 1/6 ---- -50
<> -+---- 1/6 ---- -10
     |---- 1/6 ---- 100
     `---- 1/6 ---- 250
```

This gives the Kind of the FRP representing the prize obtained. We have averaged the possibilities over different door choices.

This operator is the basis of the `evolve` built-in; see Recipe 20.

Recipe 20. Evolving a Random System

The `evolve` function is called as

```
evolve(start, next_state, n_steps=1)
```

where `start` is the Kind (or FRP) of an initial state, `next_state` is a conditional Kind (or conditional FRP) that takes in a current state and whose targets represent the next state, and `n_steps` is the number of steps to iterate over.

```
room = uniform(1, 2, 3)
move = conditional_kind({
           1: either(2, 3),
           2: constant(3)
           3: uniform(1, 2, 3)
       })
room_after_100 = evolve(room, move, 100)
```

This describes a random walk through a simple maze, where `move` represents the next room given the current room.

Recipe 21. Applying Conditional Constraints with a Statistic

We use the `|` to indicate conditional constraints, with a Kind or FRP on the left and a statistic on the right side.

```
either(1, 2) ** 3 | (Proj[2] == 1)
d_kind | Fork(Id, Sum > 4)
a_kind | And(Proj[1] > 1, Proj[2] < -1, Proj[3] == 0)
```

The parentheses are required around the statistic expresssion because `|` has low precedence. (See Recipe 23.)

Recipe 22. Applying Conditional Constraints on a Transformed Kind

Sometimes we want to apply a constraint to a transformed FRP or Kind where the constraint refers to the original Kind or FRP. This *will not work*

```
Sum(X) | (Proj[2] == 1)
```

The constraint applies to `X` but the statistic transforms it so the original information is lost.

This is the role of the `@` operator. It is a form of the transform by a statistic that *remembers* the original FRP or Kind. This *does work*

```
Sum @ X | (Proj[2] == 1)
```

Think of `Sum @ X` as an alternative form of `Sum(X)`. You could also write this as `Sum@(X)` if you prefer.

Recipe 23. Handling Operator Precedence

The operators used by `frplib` follow Python precedence rules, as described at this link. In particular, from most tightly binding to least tightly binding, the `frplib` operators are

```
[]  ()    # Indexing and evaluation/transformation
**
* @ //
>>
^
|
```

So, for example, we need parentheses around the statistic expression and the conditional constraint but no where else in

```
uniform(1, 2, 3) ** 2 ^ (Proj[1] + Proj[2])
uniform(1, 2, 3) >> prize_by_door ^ convert
a_kind | (Proj[2] > 4)
b_kind ^ Proj[4, 5, 6]
```

Recipe 24. Computing Expectations and Variances

The `E` operator computes expectations. It can be applied to a Kind, an FRP, a conditional Kind, or a conditional FRP.

```
pgd> K = uniform(1, 2, ... 11)
pgd> X = frp(K)
pgd> E(K)
6
pgd> E(X)
6
```

When applied to an FRP whose Kind is difficult to compute, `E` will use an approximation to the FRP instead. You can specify the tolerance of that approximation (optional argument `tolerance`) or even force the computation of the Kind (optional argument `force_kind`).

```
pgd> Y = frp(binary()) ** 16
pgd> E(Sum(Y))
Computing approximation (tolerance 0.01) as exact calculation may be costly
8.0266
pgd> E(Sum(Y), tolerance=0.001)
7.998249
pgd> E(Sum(Y), force_kind=True)
8
```

The last two commands take a little time.

When `E` is applied to a conditional Kind or conditional FRP, the result is a function that takes the same inputs and returns the expectation of the corresponding target.

```
pgd> prize_by_door = conditional_kind({
...>     1: either(-10, 100),
...>     2: either(-50, 250),
...>     3: constant(-100)
...> })
pgd> f = E(prize_by_door)
pgd> f(1)
45
pgd> f(2)
100
pgd> f(3)
-100
```

Similarly, the `Var` operator computes the variance of an FRP or Kind.

```
pgd> Var(uniform(-1, 0, 1))
1
pgd> Var(uniform(-10, 0, 10))
100
```

Recipe 25. Finding `frplib` Objects in Modules

The object index is included in the info system:

```
info('object-index')
```

All modules are listed with

```
info('modules')
```

Each module is automatically loaded into the playground, so you can access them with the name

```
kinds.kind
utils.dim
frps.shuffle
```

In v0.2.5+, the `frplib_objects` dictionary will show all objects imported into the playground by module.

Recipe 26. Importing `frplib` Objects in Your Code

The `frplib` modules have the form `frplib.modulename`, e.g., `frplib.kinds` and `frplib.frps`. These modules are preloaded into the playground, but in your code, you need to import them or symbols from them into your environment.

For example:

```
from frplib.frps       import frp
from frplib.kinds      import kind, uniform, weighted_as
from frplib.statistics import statistic, condition,  __, Sum, Proj
from frplib.utils      import irange, dim, codim
```

It is good practice to import only the names you need, but you can do

```
from frplib.utils      import *
```

You can also import the modules and use the names *qualified* by the module name:

```
import frplib.utils

utils.irange(1, 6)
```

Recipe 27. Using Symbolic Quantities

The functions `symbol` and `symbols` make "symbols" that can be operated on with ordinary arithmetic operators to make expressions that can be values or weights in Kinds.

```
pgd> a = symbol('a')
pgd> (1 + a) ** 2
1 + 2 a + a^2
pgd> x, y, z = symbols('x y z')
pgd> x**2 + y**2 + 2 * z**2
x^2 + y^2 + 2 z^2
pgd> u = symbols('u0 ... u9')
pgd> u[7]
u7
```

Recipe 28. Managing High-Precision Quantities

Under the hood, `frplib` uses high-precision decimals, and it converts input quantities into the correct form automatically. For this reason, one can give fractions as strings that will be converted more precisely than the built-in floats

```
weighted_as(1, 2, weights=['6/7', '1/7'])
```

The output of `frplib` functions is also in this format, and from time to time, this can cause a conflict because the high-precision decimals are not auto-convertible into standard Python floats.

```
pgd> Decimal('0.99') * 4.2
TypeError: unsupported operand type(s) for *: 'decimal.Decimal' and 'float'
```

In the rare case in which you encounter this error, you can either use `float` to convert the decimal quantity to a float or use `as_quantity` to convert the float to the right form.

```
pgd> Decimal('0.99') * as_quantity(4.2)
Decimal('4.158')
pgd> float(Decimal('0.99')) * 4.2
4.158
```

In v0.2.4+, the `as_float` utility will convert high-precision scalars and tuples to float form.

Recipe 29. Handling Tuples and Vector Tuples

For values, `frplib` uses a special type of tuple on which one can operate as vectors. These are convenient for many operations, and `frplib` does the translation for you in almost all cases.

If you want to produce such tuples, the functions `as_vec_tuple` and `vec_tuple` are helpful

```
pgd> vec_tuple(1, 2, 3)
<1, 2, 3>
pgd> as_vec_tuple([1, 2, 3])
<1, 2, 3>
pgd> as_vec_tuple(1)
<1>
pgd> vec_tuple(1)
<1>
```

Recipe 30. Getting Help

There are several ways to get help on `frplib`. Besides this Cookbook and the `frplib` Cheatsheet, Chapter 0 of the text has a wide range of examples and descriptions of how to use the tools in `frplib`. You can search in that document for examples of any particular function. In addition, the examples from the text are all accompanied by code modules which you can both use and *look at*. See for instance the `examples` directory on github.

Moreover, the playground has built-in help. You can use `help` on any function or object to see documentation and often examples. This help can be a bit "Pythony" at times, so there is also the `info` facility.

```
info()              # Index

info('overview')    # A string topic

info(weighted_as)   # associated info on an objecct
info(frp)

info('kind-factories::weighted_as')  # hierarchical topic
```

It can be useful to look at both `info` and `help`.