BART JACOBS AND KENTA CHO

EFPROB USER MANUAL

VERSION OF APRIL 24, 2018
SEE https://efprob.cs.ru.nl FOR THE LATEST INFO

Contents

Introduction

This text gives a hands-on introduction to the *EfProb* library. The latter is an embbeded language in Python, for probability. The '*Ef*' in *EfProb* stands for 'Effectus', and '*Prob* stand for 'Probability'. An effectus is an abstract (categorical) model that captures the essentials of discrete, continuous and quantum probability and logic[1]. The *EfProb* library is based on this categorical model, providing a uniform approach to discrete, continuous, and quantum probability. However, in order to be able to use and understand the basic of the *EfProb* library it is not required to understand the underlying categorical semantics.

[1] B. Jacobs. New directions in categorical logic, for classical, probabilistic and quantum logic. *Logical Methods in Comp. Sci.*, 11(3):1–76, 2015; and K. Cho, B. Jacobs, A. Westerbaan, and B. Westerbaan. An introduction to effectus theory. see arxiv.org/abs/1512.05813, 2015

The *EfProb* library makes it easy to model various problems in probability theory and to compute outcomes: probabilities of events and of predicates can be computed, product and conditional states can be formed, expectations and variances can be calculated, *etc.* These computations can be done for *e.g.* Bayesian networks, hidden Markov models, stochastic kernels, and quantum protocols.

A short introduction to *EfProb* has been published.[2] It can be used to get a quick overview. This manual gives a more extensive hands-on introduction.

[2] K. Cho and B. Jacobs. The EfProb library for probabilistic calculations. In F. Bonchi and B. König, editors, *Conference on Algebra and Coalgebra in Computer Science (CALCO 2017)*, volume 72 of *LIPIcs*. Schloss Dagstuhl, 2017

The *EfProb* library consists of two Python files:

1. `efprob_dc.py`, for discrete and continuous probability, in combined form, discused in Chapters 1 and 2;

2. `efprob_qu.py`, for quantum probability, described in Chapter 3.

There is some overlap in the explanations in the three chapters. This is intentional, so that these chapters can be read largely independently. The reader is encouraged to load these files before studying the relevant chapters, and to try out the examples and make variations on them while progressing through the text.

In the text below many examples are shown using Python in interactive mode, from the command line, as in:

```
>>> from efprob_dc import *
>>> s = flip(0.2)
```

```
>>> s
0.2|True> + 0.8|False>
```

The lines starting with >>> describe user input, and the other lines describe the output given by the system. The first line imports the relevant library. It will be omitted in the sequel.

All the examples used in this manual are also collected in three separate files, with minimal explanations. These files have self-explanatory names:

- `discrete-illustrations.py`
- `continuous-illustrations.py`
- `quantum-illustrations.py`

These files can be excuted, modified, and studied more closely.

The basics of the *EfProb* library exist and are reasonably stable. This basis is already being used in several research papers. This usage is the reason for publicly releasing the library at this early stage, so that results can be reconstructed and checked by others. Further research on extensions and adaptations of the basic library are still going on. Therefore new versions will appear from time to time.

For the same reason, this manual is still evolving, and is incomplete as it stands. Still we hope that it is useful, because it already explains many basic aspects of the library and contains several examples of how it can be used.

We conclude this introduction with several loosely related remarks about the *EfProb* library.

- The focus of the development of the library is on providing a new language and logic for probability, based on states and predicates. The library is not meant for large scale computation, for instance like in data analytics. In our development of the library we prefer semantical clarity to execution speed. Also, the *EfProb* library does not come with any guarantees with respect to the correctness of the outcomes, see also the discussion about precision and side-conditions below.
- We envision that the library could be useful in *teaching* the basics of probability theory from a unified perspective. The *EfProb* library is especially useful in quickly calculating probabilities, and experimenting with various formulations.
- At the same time the library could be useful in scientific *research* as well, in order to quickly model new examples and compute outcomes. Many researchers in (classical and quantum) probability

have developed their own scripts for quickly calculating probabilities. The *EfProb* library could develop into a common language in which results can be formulated uniformly, so that they can be verified, compared or re-used by colleagues. We request that usage of the *EfProb* library, if any, is mentioned in the acknowledgements of research papers.

- The *EfProb* library requires Python, version ≥ 3.5. *EfProb* heavily relies on existing libraries in Python, in particular `math` and `numpy` (used as `np`) for linear algebra and matrices.

- It is important to keep in mind that the main functionality of the library is *computing* numerical outcomes, and not, for instance, logical reasoning. Often, these numerical outcomes are approximations, arising from the inherent limitations of using a computer for real number arithmetic. Sometimes we do use random states and/or random predicates to *test* certain properties. But these numerical tests should never be taken as a logical *proof*.

- Despite the fact that numerical approximations are used in the EfProb calculations, the underlying semantics is exact, and the library computes probabilities using standard mathematical formulas, including, for instance, integration (which is approximate, unlike[3], which uses exact symbolic integration). Still, this is in contrast to language based on sampling, such as BLOG[4] and Church[5].

- Many mathematical structures used in probability theory are required to satisfy certain properties. For instance: the probabilities occurring in a distribution must add up to one; or: the integral over the domain of definition of a probability density function (pdf) must be one; or: a quantum state is a matrix that must be positive and have a trace that is equal to one.

  A fundamental question that exists in the current setting is: should such properties be checked by *EfProb* or should it be left as a responsability of the user that these properties are satisfied — for instance when defining distributions / pdf's / quantum states?

  Here it is relevant that in the presence of computational approximations equations — such as: a trace equals one — can never be checked reliable. In the continuous case the problem is more serious, since it must be checked that a certain property (*e.g.* positivity of a function) holds for infinitely many points, and it's not effectively possible to do so. Hence, necessarily, the responsability for side-conditions lies with the user.

  That being said, the *EfProb* does perform a few basic checks, sometimes using margins of error. In our experience, these checks can be useful to detect and prevent mistakes. Some check func-

[3] T. Gehr, S. Misailovic, and M. Vechev. PSI: Exact symbolic inference for probabilistic programs. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification*, number 9779 in Lect. Notes Comp. Sci., pages 62–83. Springer, Berlin, 2016

[4] B. Milch, B. Marthi, S. Russell, D. Sontag, D. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In L. Getoor and B. Taskar, editors, *Statistical Relational Learning*. MIT Press, 2007

[5] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008

tions only print warnings, explaining the margins, when side-conditions fail. The checking of side-conditions is thus handled pragmatically. It is one of the least stable parts of the library.

- Because side-conditions are not checked systematically, the *EfProb* language is unsafe. What we have is the familiar tension between expressive embedded languages and denotationally-sound type-safe languages. For instance, when something goes wrong in *EfProb*, the error message typically comes from Python; one needs to understand the embedding of *EfProb* in Python to some extent in order to translate the error back to *EfProb*.

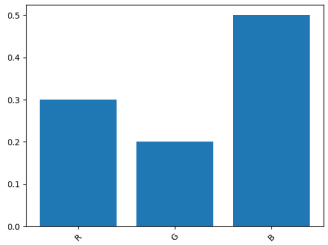
  But again, the *EfProb* library does perform some basic checks. For instance, states, predicates and channels use 'domains' as types, as will be explained below. It is checked that types match, for instance when the validity of a predicate in a state is computed, or when state/predicate transformation is applied. In case of a non-match, an exception is thrown.

1
Discrete Probability

A *discrete finite probability distribution*, or simply a *distribution* for short, is a weighted combination of 'things', where the weights are numbers from the unit interval $[0,1]$ that add up to one. Such a weighted sum is also called a *convex combination*. Here is an example, using the RGB colour model. This model describes each colour as an additive combination of the primary colours red (R), green (G) and blue (B). A particular colour can then be described as a convex combination:

$$0.3|R\rangle + 0.2|G\rangle + 0.5|B\rangle.$$

This formal sum describes a particular weighted combination of red, green, and blue. The 'ket' notation $|-\rangle$ is borrowed from the quantum world. It has no mathematical meaning, but is convenient to separate the weights in the unit interval $[0,1]$ from the 'things' that are weighed. Such distributions are sometimes called 'categorical', but here we avoid this terminology since it is confusing in a context where category theory plays an important role. A graphical representation ('plot') of the above convex combination can be seen in the margin.



This chapter describes how to form such distributions, and how to manipulate them via various other structures, like predicate, channels and random variables. The same approach will be used for continuous probability and also for quantum probability, so that the similarities (and also some dissimilarities) between the three forms of probability become clear. In order to emphasise this similarity, we will use uniform terminology. Therefore, we often call a (discrete finite probability) distribution a *state*.

1.1 States

In our embedded language *EfProb* within Python one can define a state for a fair (balanced) coin as:

```
>>> fflip = State([0.5, 0.5], [True, False])
>>> fflip
0.5|True> + 0.5|False>
```

We see that the 'things' that we weigh in this distribution are in the second list argument of the class constructor `State`. The set of these 'things' is what we call the *domain* of the distribution. The elements of the domain of a distribution are printed using the ket-notation $|-\rangle$, as explained in the beginning of this chapter. Subsection 1.1.2 below describes how distributions can be plotted.

We can also define use another domain, as in:

```
>>> fflipHT = State([0.5, 0.5], ['H', 'T'])
>>> fflipHT
0.5|H> + 0.5|T>
```

In principle, anything is allowed as domain element, as long as it is already known to Python at the moment that the state is defined.[1]

[1] One can define one's own domain elements in Python, but this should be done before the definition of the state that uses these domain elements.

Since the booleans `True` and `False` are often combined in a domain, there is a predefined boolean domain for them:

```
bool_dom = Dom([True, False])
```

EfProb is an embedded language within Python. This means that we can define Python itself to define a parametrised coin as:

```
>>> def flip(r): return State([r, 1-r], bool_dom)
>>> flip(0.2)
0.2|True> + 0.8|False>
```

This flip function is actually predefined in *EfProb*.

In a similar way we can now define a fair dice as:

```
>>> fdice = State([1/6,1/6,1/6,1/6,1/6,1/6], [1,2,3,4,5,6])
>>> fdice
0.167|1> + 0.167|2> + 0.167|3> + 0.167|4> + 0.167|5> + 0.167|6>
```

The *EfProb* library does not check or enforce that the probabilities in user-defined states add up to one. As explained in the introduction, the main reason is that such checks have limited value because of small precision errors in calculations in Python. In addition, it gives some more flexibility, for instance to use *subdistribution*, where the sum of probabilities is below one. The disadvantage of not having these checks is that non-probabilistic and even false results can be obtained.

The probabilities are pretty-printed with only three decimals. This can be adjusted in the library itself.

There is a predefined function for uniform distributions, for which we only have to give the domain. Hence it is easier to define:

```
>>> fdice = uniform_state([1,2,3,4,5,6])
>>> fdice
0.167|1> + 0.167|2> + 0.167|3> + 0.167|4> + 0.167|5> + 0.167|6>
```

If you like, you can also use a sequence of points as domain:

```
>>> fdice = uniform_state(['*','**','***','****','*****','******'])
>>> fdice
0.167|*> + 0.167|**> + 0.167|***> + 0.167|****> + 0.167|*****> + 0.167|******>
```

There is also a function that produces a singleton state $1|x\rangle$ with domain D (which must contain x), namely `point_state(x, D)`, see for instance:

```
>>> point_state(True, bool_dom)
1|True> + 0|False>
>>> point_state(3, [1,2,3,4,5,6])
0|1> + 0|2> + 1|3> + 0|4> + 0|5> + 0|6>
```

When you are happy with natural numbers as domain you can use Python's built-in function `range`. For an arbitrary number n, `range(n)` is the sequence of $0, 1, 2, \ldots, n-1$ and can be used as domain:

```
>>> uniform_state(range(5))
0.2|0> + 0.2|1> + 0.2|2> + 0.2|3> + 0.2|4>
>>> point_state(2, range(4))
0|0> + 0|1> + 1|2> + 0|3>
```

The simplest possible distribution $1|0\rangle$ with domain $\{0\}$ can be produced as `point_state(0,range(1))` or also as `uniform_state(range(1))`.

For testing purposes it is useful to have a random discrete probability. It works as described below, internally using Python's random number generator. Each time it gives a different answer.

```
>>> random_state(range(5))
0.133|0> + 0.273|1> + 0.175|2> + 0.119|3> + 0.3|4>
>>> random_state(range(5))
0.252|0> + 0.25|1> + 0.452|2> + 0.035|3> + 0.0103|4>
```

The binomial distribution with parameters $N \in \mathbb{N}$ and $p \in [0,1]$ describes for each $k \in \{0, 1, \ldots, N\}$ the probability of getting k successes out of N attempts. Formally, this distribution is given by the formal convex sum:

$$\sum_{0 \leq k \leq N} \binom{N}{k} \cdot p^k \cdot (1-p)^{N-k} |k\rangle.$$

These binomial distributions are predefined as a discrete (parametrised) state in *EfProb*, giving for instance:

```
>>> binomial(6, 0.3)
0.118|0> + 0.303|1> + 0.324|2> + 0.185|3> + 0.0595|4> + 0.0102|5> + 0.000729|6>
```

Its domain is `range(N+1)`.

Remark 1 *As sketched above, several basic* finite *discrete distributions can be represented in EfProb. But what about* infinite *discrete distributions? The most prominent example is the* Poisson *distribution, with the natural numbers* $\mathbb{N}$ *as domain, given by the formal sum:*

$$\sum_{k \in \mathbb{N}} \frac{\lambda^k \cdot e^{-\lambda}}{k!} |k\rangle$$

where $\lambda \geq 0$ is the 'mean' or 'rate' parameter.

The representation of the Poisson distribution/state in EfProb involves restriction to a certain upper bound ub $\in \mathbb{N}$*, making the above distribution finite. But since the weights then no longer add up to one, we compensate in an appropriate way. This representation is thus an approximation of the 'real' Poisson distribution, where the user can choose the upper bound sufficiently high for the required level of precision.*

The implementation of this restricted Poisson distribution looks as follows.

```
def poisson(lam, ub):
    probabilities = [(lam ** k) * (e ** -lam) / factorial(k)
                     for k in range(ub)]
    s = sum(probabilities)
    return State([p/s for p in probabilities], range(ub))
```

With mean 3 *and upperbound* 20 *we see that the last probability is already really small:*

```
>>> poisson(3,20)
0.0498|0> + 0.149|1> + 0.224|2> + 0.224|3> + ... + 3.01e-09|18> + 4.76e-10|19>
```

1.1.1 Operations on states

The three standard operations on states are: product, convex sum, and marginal. We discuss them one by one below.

Product states A 'joint' or 'product' distribution/state can be formed via the product operator `@`. For instance, the product of a coin with bias `0.2` and a coin with bias `0.7` is written as:

```
>>> flip(0.2)
0.2|True> + 0.8|False>
>>> flip(0.7)
0.7|True> + 0.3|False>
>>> flip(0.2) @ flip(0.7)
0.14|True,True> + 0.06|True,False> + 0.56|False,True> + 0.24|False,False>
```

We see that the product distribution contains all possible pairs of domain elements, with multiplied probabilities. Of course, the sum of the weights is still one.

It is is not necessary that the two states that are combined in a product have the same length or domain:

```
>>> flip(0.2) @ uniform_state(range(4))
0.05|True,0> + 0.05|True,1> + 0.05|True,2> + 0.05|True,3> +
   0.2|False,0> + 0.2|False,1> + 0.2|False,2> + 0.2|False,3>
```

Multiple products are also possible:

```
>>> flip(0.2) @ uniform_state(range(4)) @ flip(0.8)
0.04|True,0,True> + 0.01|True,0,False> + 0.04|True,1,True> + 0.01|True,1,False> +
   0.04|True,2,True> + 0.01|True,2,False> + 0.04|True,3,True> +
   0.01|True,3,False> + 0.16|False,0,True> + 0.04|False,0,False> +
   0.16|False,1,True> + 0.04|False,1,False> + 0.16|False,2,True> +
   0.04|False,2,False> + 0.16|False,3,True> + 0.04|False,3,False>
```

Since the domain of a product state is the cartesian product of the domains of the component states, such products states grow quickly in size. If we have k states $s_1, s_2, \ldots, s_k$, where each state s_i has domain size n_i, then the product state $s_1 \mathbin{@} s_2 \mathbin{@} \cdots \mathbin{@} s_k$ has size $n_1 \cdot n_2 \cdot \ldots \cdot n_k$.[2]

One can use Python's power operator `**` to get multiple products of a state with itself:

[2] The parallel product operator `@` is 'strictly' associative, so that `(s1 @ s2) @ s3` and `s1 @ (s2 @ s3)` are the same states.

```
>>> flip(0.2) ** 3
0.008|True,True,True> + 0.032|True,True,False> + 0.032|True,False,True> +
   0.128|True,False,False> + 0.032|False,True,True> + 0.128|False,True,False> +
   0.128|False,False,True> + 0.512|False,False,False>
```

This is the same as `flip(0.2) @ flip(0.2) @ flip(0.2)`.

Convex sums of states Discrete states are formal convex sums over their domains. But there is an additional form of convex sum, namely of states themselves. For instance in:

```
>>> convex_sum( [ (0.2,flip(0.3)), (0.5,flip(0.8)), (0.3,flip(1)) ] )
0.76|True> + 0.24|False>
```

In such a convex sum of states it is required that the weights (here: `0.2`, `0.5` and `0.3`) add up to one and that all the states have the same domain.

Marginalisation of joint states The product operation `@` *constructs* a new, joint state from 'component' states. Marginalisation *destructs* a joint state. When applied to a state built with `@`, marginalisation returns the original components. In order to give a bit more generality to a demonstration like the one below, we use randomly generated states.

```
>>> s = random_state(range(3))
>>> t = random_state(range(2))
>>> s
0.255|0> + 0.465|1> + 0.28|2>
>>> t
0.507|0> + 0.493|1>
>>> s @ t
0.13|0,0> + 0.126|0,1> + 0.236|1,0> + 0.229|1,1> + 0.142|2,0> + 0.138|2,1>
>>> (s @ t) % [1,0]
0.255|0> + 0.465|1> + 0.28|2>
```

```
>>> (s @ t) % [0,1]
0.507|0> + 0.493|1>
```

We use the post-fix operation `% [1,0]` for the *first* marginal, and `% [0,1]` for the *second* marginal. On a state built up from n components one can use 'selectors' given by a list of n `0`'s and `1`'s. A '`1`' at position i means: keep this i-th component; a '`0`' at position i says: delete this i-th component. For instance we remove the really big state and keep the two smaller ones in:

```
>>> (random_state(range(100)) @ flip(0.5) @ uniform_state(range(2))) % [0,1,1]
0.25|True,0> + 0.25|True,1> + 0.25|False,0> + 0.25|False,1>
```

Marginalisation involves summation over all probabilities that are projected out, *i.e.* that are discarded in a certain dimension, as indicated by a `0` in a marginalisation selector `% [...]`.

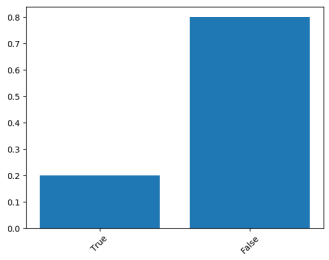
In general, a joint state `s` is called *independent* or *non-entwined* if it is the product of its marginals, that is, if it is equal to `(s % [1,0]) @ (s % [0,1])`. If the state `s` is defined as product `s1 @ s2`, then it is always non-entwined. We shall see an illustration of an entwined state (via conditioning) in Example 5.

We add the following general observation about entwinedness of joint states.

$$r_1|0,0\rangle + r_2|0,1\rangle + r_3|1,0\rangle + r_4|1,1\rangle \quad \text{is non-entwined} \quad \text{iff} \quad r_1 \cdot r_4 = r_2 \cdot r_3 \tag{1.1}$$

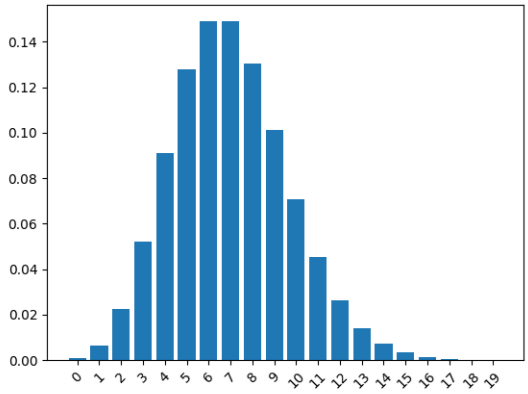
1.1.2 *Plotting states*

So far we have described states as finite formal convex sums $r_1|x_1\rangle + \cdots + r_n|x_n\rangle$. The *EfProb* library also supports, to some extent, plotting states, as bar charts. The command below generates the picture in the margin.

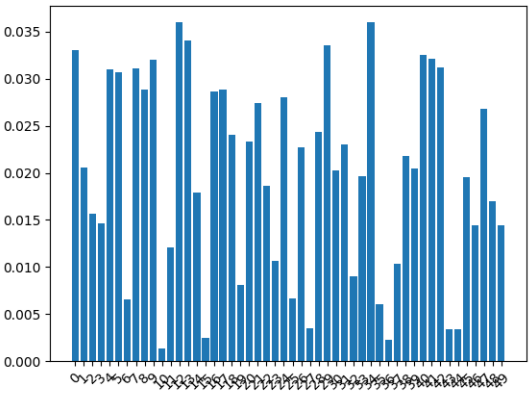



```
>>> flip(0.2).plot()
```

The probability of each element of the domain is indicated by the height of the corresponding bar. This works for domains of a 'reasonable' size. For instance, the plot of the the state `poisson(7,20)` gives the picture below.



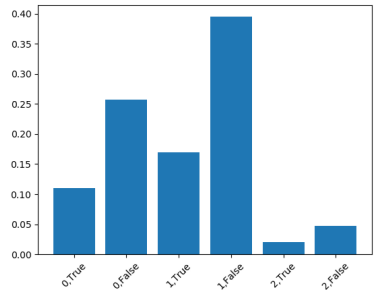
This is still quite readable. But when we move to a domain of 50 elements the plot becomes quite dense and the individual domain elements are hard to distinguish. The example below describes an instance of `random_disc_state(50)`.

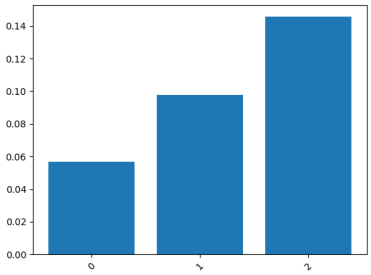


A 2-dimensional product/joint state can also be plotted. In addition, one can plot a 'slice' of a product state, as shown below.

```
>>> s = State([0.1,0.5,0.25,0.15], range(4))
>>> (s @ flip(0.3)).plot()
>>> (s @ flip(0.3)).plot(...,True)
```

This gives, respectively:





1.1.3 *An excursion on domains of states*

The domain of distribution is a list of 'things' that can be made visible via the `dom` attribute of a state:

```
>>> flip(0.8).dom
[True, False]
```

This boolean domain exists in *EfProb* as `bool_dom`. If we ask for the domain of a product state we get the combination of their domains, printed via $*$ as:

```
>>> (flip(0.8) @ uniform_state(range(5))).dom
[True, False] * range(0, 5)
```

You can go one level deeper and ask for the discrete domain; it yields a list of domains:

```
>>> (flip(0.8) @ uniform_state(range(5))).dom.disc
[[True, False], range(0, 5)]
```

So far we have constructed joint states only via the product operator `@`. But we can also defined them directly, as in:

```
>>> s = State([0.1,0.1,0.1,0.7], [[True,False], [0,1]])
>>> s
0.1|True,0> + 0.1|True,1> + 0.1|False,0> + 0.7|False,1>
```

This is relevant especially if we wish to define a non-entwined state.

1.2 *Predicates*

Predicates in logic are statements about some space/domain of elements, such as $x \geq 5$, where x is a variable that ranges for instance over the domain of natural or real numbers. In predicate logic predicates are standardly interpreted as *subsets* $S \subseteq X$ of a given space X. In the present context we call such predicates *sharp*. They can be identified with their 'characteristic' function $\mathbf{1}_S \colon X \to \{0,1\}$ given by

$\mathbf{1}_S(x) = 1$ iff $x \in S$. Here, in probabilistic logic, we shall use a more general notion of predicate, given by a function $p\colon X \to [0,1]$, where $[0,1] \subseteq \mathbb{R}$ is the unit interval. Sharp predicate are obviously included in this more general 'fuzzy' notion of predicate. In a probabilistic setting a sharp predicate is often called an *event*.

The collection of sharp predicates on a set X has a certain algebraic/logical structure, captured by the notion of a Boolean algebra. There are truth and falsity predicate $\mathbf{1}, \mathbf{0}, X \to \{0,1\}$ given by the functions which are always 1, or always 0. Also there is negation, written as $\sim p$, and defined by $(\sim p)(x) = 1 - p(x)$. Clearly, $\sim\sim p = p$, $\sim\mathbf{1} = \mathbf{0}$ and $\sim\mathbf{0} = \mathbf{1}$. In addition, there is conjuction of predicates $p \wedge q$ given by pointwise multiplication: $(p \wedge q)(x) = p(x) \cdot q(x)$.

These logical operations do not only work for sharp predicate $X \to \{0,1\}$ but for non-sharp $X \to [0,1]$ as well. But in that case we do not get a Boolean algebra, but what is called an *effect module*[3]. Here it does not directly matter what the general laws of an effect module are; it suffices to know the relevant operations on predicates $X \to [0,1]$. Besides negation $\sim$, truth and falsity, there are two more operation in an effect module that we have not yet mentioned, namely:

[3] B. Jacobs. New directions in categorical logic, for classical, probabilistic and quantum logic. *Logical Methods in Comp. Sci.*, 11(3):1–76, 2015

- partial sum $p + q$ of predicates p, q; it is defined if $p(x) + q(x) \leq 1$, for all $x \in [0,1]$, and in that case it is determined as pointwise addition: $(p+q)(x) = p(x) + q(x)$;
- scalar multiplication $r \cdot p$, where $r \in [0,1]$ is a 'probabilistic scalar' and p is a predicate. The resulting predicate $r \cdot p\colon X \to [0,1]$ is defined by $(r \cdot p)(x) = r \cdot p(x)$.

Recall that a distribution on a (finite) set X can be identified with a 'mass' function $\omega\colon X \to [0,1]$ that satisfies $\sum_x \omega(x) = 1$. Thus, a state is a predicate, but not the other way around. A predicate is also a function $p\colon X \to [0,1]$, but for predicates there is no requirement that the probabilities $p(x) \in [0,1]$ add up to one. Although they look similar, it is important to keep states and predicates apart. Predicates are closed under certain operations, like scalar multiplication, which do not exist for states.

In the sequel we shall use two important operations combining a state ω and a predicate p, namely *validity* $\omega \models p$, and conditioning $\omega|_p$. But first we describe the basic 'effect module' operations on predicates in the *EfProb* library.

1.2.1 Operations on predicates

In this section we discuss the following operations on predicates.

- orthosupplement, or negation $\sim$

- scalar multiplication `*`
- partial sum `+`
- parallel conjunction `@`
- sequential conjunction `&`

But first we mention a few ways to obtain predicates in the first place.

Predicates in *EfProb* have a domain, exactly like states have a domain. We sometimes say that `p` is a predicate on state `s` if `p` and `s` have the same domain. This will be most relevant when we discuss validity and conditioning later on, but it is good to have this idea already in mind now.

A (discrete) predicate is given by a domain together with a probability in $[0,1]$ for each of the domain elements. These probabilities are listed first, as in:

```
>>> p = Predicate([0.2, 1, 0.5, 0.0], ['a', 'b', 'c', 'd'])
>>> p
a: 0.2 | b: 1 | c: 0.5 | d: 0
```

When printed, these domain elements are enumerated together with the associated probability.

There are several predefined predicates. The truth predicate on the boolean domain `[True,False]` is described as:

```
>>> truth(bool_dom)
True: 1 | False: 1
```

Similarly, we can describe for instance the falsity predicate on the domain of the dice distribution:

```
>>> falsity([1,2,3,4,5,6])
1: 0 | 2: 0 | 3: 0 | 4: 0 | 5: 0 | 6: 0
```

It is useful to have a predicate which is 1 only at one element of its domain. This predicate is called `point_pred`. It is illustrated in:

```
>>> point_pred(3, range(4))
0: 0 | 1: 0 | 2: 0 | 3: 1
```

There are predefined boolean point predicates which are often convenient:

```
yes_pred = point_pred(True, bool_dom)
no_pred = point_pred(False, bool_dom)
```

In the previous section we have seen a function `random_state`. There is a similar random-predicate function — but this time the probabilities do not add up to one.

```
>>> random_pred(range(5))
0: 0.806 | 1: 0.593 | 2: 0.758 | 3: 0.68 | 4: 0.442
```

Each time this function is called, different probabilities appear.

As mentioned in the beginning of this section, each state is a predicate — but not the other way around. In the *EfProb* library the casting of a state to a predicate can be done via an explicit method on states, called `as_pred`.

```
>>> s = flip(0.75)
>>> s
0.75|True> + 0.25|False>
>>> s.as_pred()
True: 0.75 | False: 0.25
```

We shall make only limited use of this `as_pred` method. In our experience it works well to define predicates directly, instead of via states.

In the remainder of this section we describe operations on predicates.

Orthosupplement As described in the beginning of this section, the orthosupplement of a predicate p changes all of its probabilities $p(x)$ into $1 - p(x)$, for each element x in the domain, see:

```
>>> p = random_pred(range(5))
>>> p
0: 0.15 | 1: 0.803 | 2: 0.132 | 3: 0.819 | 4: 0.855
>>> ~p
0: 0.85 | 1: 0.197 | 2: 0.868 | 3: 0.181 | 4: 0.145
```

Obviously, the orthosupplement of truth is falsity; and taking the orthosupplement twice returns the original predicate.

Scalar multiplication Each predicate p can be multiplied pointwise by a scalar $r \in [0, 1]$, as in:

```
>>> s = flip(0.75)
>>> p = s.as_pred()
>>> p
True: 0.75 | False: 0.25
>>> 0.2 * p
True: 0.15 | False: 0.05
```

We see that the result of this scalar multiplicaiton is no longer a state. There are some obvious equalities, making $*$ a monoid action, namely $1 * p = p$ and $r_1 * (r_2 * p) = (r_1 * r_2) * p$. In addition, scalar multiplication with zero yields falsity.

In this way we can define a 'uniform' predicate, says with domain `range(6)` and constant value `0.3`, as:

```
>>> 0.3 * truth(range(6))
0: 0.3 | 1: 0.3 | 2: 0.3 | 3: 0.3 | 4: 0.3 | 5: 0.3
```

Partial sum Pointwise addition of predicates p, q is only possible when $p(x) + q(x) \leq 1$ for each element x of the domain.[4] In that case we can safely add predicates p, q to form a partial sum predicate $p + q$, as in:

[4] This condition is not checked in the *EfProb* library.

```
>>> point_pred(3,range(4)) + point_pred(0,range(4))
0: 1 | 1: 0 | 2: 0 | 3: 1
```

The sum of predicates `p` and `~p` always exists and yields the (constant 1) truth predicate.

It is not hard to see that predicates, like states are closed under convex combinations. With the sum operator `+` we can write such combinations simply as:

```
>>> 0.3 * truth(range(4)) + 0.7 * point_pred(0,range(4))
0: 1 | 1: 0.3 | 2: 0.3 | 3: 0.3
```

Parallel conjunction Recall from Subsection 1.1.1 that states can be put in parallel with the operator `@`. Similarly, if we have predicates `p1` on state `s1` and `p2` on state `s2`, then `p1 @ p2` is a predicate on `s1 @ s2`. Below is an example of such a parallel conjunction predicate predicate, together with its domain.

```
>>> p = Predicate([0.2, 0.8, 0.5, 0.0], ['a', 'b', 'c', 'd'])
>>> q = truth(bool_dom)
>>> p @ q
a,True: 0.2 | a,False: 0.2 | b,True: 0.8 | b,False: 0.8 |
   c,True: 0.5 | c,False: 0.5 | d,True: 0 | d,False: 0
>>> (p @ q).dom
['a', 'b', 'c', 'd'] * [True, False]
```

Thus, the domain is the cartesian product of the underlying domains; and the probability on a tuple in this cartesian product is the product of the probabilities on the tuples. For this reason, the following predicates are the same:

```
(r * p) @ q      r * (p @ q)      p @ (r * q)
```

If the sum `p1 + p2` is defined, the following predicates are also the same.

```
(p1 + p2) @ q      (p1 @ q) + (p2 @ q)
```

Similarly, `@` preserves `+` in its second argument.

Sequential conjunction In contrast to parallel conjunction `@`, sequential conjunction `&` requires that the two predicates involved have the same domain. The result is the obtained by the pointwise multiplication of the probabilities on each domain element:

```
>>> p1 = Predicate([0.3, 0.8, 0.01], ['x', 'y', 'z'])
>>> p2 = Predicate([0.5, 0.2, 0.25], ['x', 'y', 'z'])
>>> p1 & p2
x: 0.15 | y: 0.16 | z: 0.0025
>>> (p1 & p2).dom
['x', 'y', 'z']
```

This conjunction operator `&` is commutative and has truth as neutral element. It preserves sums `+` in each argument separately.

Using sequential conjunction `&` we can define disjunction `|` as its De Morgan dual:

`p | q` is `~( ~p & ~q )`

In quantum logic, this operator & is *not* commutative and corresponds to 'and-then'; there it really makes sense to call & *sequential* conjunction, see Subsection 3.2.1. In the current setting of discrete (and continuous) probability we use the same name for consistency. An additional advantage of calling & 'sequential conjunction' — instead of just 'conjunction' — is that this name is clearly different from 'parallel conjunction' `@`.

1.2.2 *Validity*

For a state $\omega = \sum_i r_i|x_i\rangle$ with domain X and a predicate $p\colon X \to [0,1]$ one can define the validity $\omega \models p$ of predicate p in state ω. This validity $\omega \models p$ is a number in the unit interval $[0,1]$ defined as:

$$\omega \models p \;=\; \sum_i r_i \cdot p(x_i). \tag{1.2}$$

This number describes the expected value of p in ω.

In Python the operator `|=` already has a fixed meaning. We shall use instead `>=` for validity, since it looks reasonably similar to $\models$. Let a state `s` have a domain with n elements, with corresponding probabilities s_i, for $0 \leq i < n$, where $\sum_i s_i = 1$. Similarly, let a predicate `p` have the same domain with n elements, with probabilities p_i. The validity `s >= p` is then given by the formula $\sum_i s_i \cdot p_i$. More abstractly, it is the matrix product $p^T \cdot s$, where p^T is the transpose of p.

Here is an example, using the (fair) dice state `fdice` from Section 1.1 and the predicates 'even' and 'odd', telling whether the number of dots is even or odd.

```
>>> fdice = uniform_state([1,2,3,4,5,6])
>>> even = Predicate([0,1,0,1,0,1], [1,2,3,4,5,6]))
>>> odd = ~even
>>> fdice >= even
0.5
>>> fdice >= odd
0.5
>>> fdice >= even + odd
1.0
>>> fdice >= even & odd
```

```
0.0
>>> fdice >= even | odd
1.0
```

We see that sum `+` and disjunction `|` give the same outcome. This is because the predicates `even` and `odd` are sharp. In general there is a difference, see *e.g.*:

```
>>> fdice >= (0.5 * even + 0.2 * odd) + even
0.85
>>> fdice >= (0.5 * even + 0.2 * odd) | even
0.6
```

The message is that one should be careful in reading the logical operators `+` and `|` as 'or' in a probabilistic setting.

Validity $\models$ satisfies some basic mathematical properties:

$$\begin{aligned}
\omega \models \mathsf{truth} &= 1 \\
\omega \models \mathsf{falsity} &= 0 \\
\omega \models {\sim} p &= 1 - (\omega \models p) \\
\omega \models r * p &= r * (\omega \models p) \\
\omega \models p_1 + p_2 &= (\omega \models p_1) + (\omega \models p_2) \\
\omega \mathbin{@} \sigma \models p \mathbin{@} q &= (\omega \models p) \cdot (\sigma \models q).
\end{aligned}$$

Such properties can be tested (but not proven!) by using random states and predicates, as in:

```
>>> s1 = random_state(range(100))
>>> s2 = random_state(range(50))
>>> p1 = random_pred(range(100))
>>> p2 = random_pred(range(50))
>>> (s1 @ s2) >= (p1 @ p2)
0.25646568531521891
>>> (s1 >= p1) * (s2 >= p2)
0.25646568531521891
```

The validity `s >= p` is defined only when the state `s` and the predicate `p` have the same domain. Often it happens that the domain of the state is 'broader' than the domain of the predicate. In that case we need to 'widen' the predicate. This is called *weakening*.

More generally, if you have defined a term or a predicate in a context of assumptions Γ, then you can also use this same term or predicate in an enlarged context Γ, Δ. Often, weakening is done implicitly. However, in certain settings weakening requires more care.

1. In a *linear* setting, like in linear logic, weakening does not exist. In the current setting of quantum logic, we do have weakening, since discarding is possible in a quantum setting.[5]

[5] The companion of weakening is *contraction*, making it possible to move someting from a context Γ, Γ to a context Γ. This involves copying of resources, which is not possible in a quantum setting, because of no-cloning. Semantically, weakening requires projections $\Gamma, \Delta \to \Gamma$, whereas contraction requires diagonals $\Gamma \to \Gamma, \Gamma$.

2. Usually in logic, weakening is handled automatically, via variables, see the the next illustration.

$$\frac{x : A \vdash M(x) : B}{x : A, y : C \vdash M(x) : B}$$

The term $M(x)$, depending on variable $x : A$, is weakened to a larger context with an additional variable $y : B$. In doing so the term M does not change. This may seem obvious, but in a formalism without variables, like in the current embedded language *EfProb*, we need to make clear somehow that M has moved.

Here is an example.

```
>>> fdice = uniform_state([1,2,3,4,5,6])
>>> even = Predicate([0,1,0,1,0,1], [1,2,3,4,5,6])
>>> fdice >= even
0.5
>>> fdice @ flip(0.3) >= even @ truth(bool_dom)
0.5
>>> flip(0.3) @ fdice >= truth(bool_dom) @ even
0.5
```

The predicate `even` can be used as a predicate on the state `fdice`, but not on the 'broader' states `fdice @ flip(0.3)` or on `flip(0.3) @ fdice`. However, the predicate `even` can be widened to these product states, via parallel conjunction `@` with `truth`, on the right or on the left. Notice that we do have to provide the `truth` predicate with the right domain.

Weakening involves some bookkeeping, see for instance Example 4 below. It is not difficult, but one has to be aware of it, in order to avoid error messages. The conclusion is:

> Weakening of a predicate works via parallel conjunction with truth, either on the left or on the right. One then extends the type of a predicate so that it matches a larger state, without affecting the validity of the predicate.

1.2.3 *Conditioning*

Conditional probability deals with probabilities 'given some event'. In effectus theory and in the *EfProb* library one can condition a state with a predicate. We start with an illustration, using the familiar fair dice:

```
>>> fdice = uniform_state([1,2,3,4,5,6])
>>> even = Predicate([0,1,0,1,0,1], [1,2,3,4,5,6])
>>> atmost4 = Predicate([1,1,1,1,0,0], [1,2,3,4,5,6])
>>> fdice >= even
```

```
0.5
>>> fdice >= atmost4
0.66666666666666667
```

Now suppose that we wish to know the probability of `even`, given that the outcome is at most 4. This is obtained as:

```
>>> fdice / atmost4 >= even
0.5
```

The state `fdice / atmost4` is a *conditional* state, obtained from `fdice` via conditioning. The probabilities in `fdice` are updated (revised) in the light of the predicate `atmost4`. Explicitly,

```
>>> fdice / atmost4
0.25|1> + 0.25|2> + 0.25|3> + 0.25|4> + 0|5> + 0|6>
```

We see that the domain elements `5` and `6` have probability `0` — since the predicate `atmost4` is false for `5` and `6` — and that the probabilities of the other elements — for which `atmost4` does hold — have been normalised, so that they add up to one. It may be clear that in this conditional, updated state `fdice / atmost4` the probability of getting an even number of dots is $\frac{1}{2}$.

Similarly:

```
>>> fdice / even
0|1> + 0.333|2> + 0|3> + 0.333|4> + 0|5> + 0.333|6>
>>> fdice / even >= atmost4
0.66666666666666667
```

Remark 2 *We make the difference explicit between the traditional notation used in conditional probability (on the left below), and the one that we use (on the right).*

$$\Pr(\texttt{even | atmost4}) \qquad \textit{and} \qquad \texttt{fdice / atmost4 >= even}$$

We note the following.

1. *In our notation the underlying state (*`fdice`*) is written explicitly, whereas it is left implicit in the traditional notation.*

2. *Our notation explicitly writes the modification of this state. The conditioned state* `fdice / atmost4` *is useful in itself, for instance in conditional expectation, see Section 1.3.*

3. *The traditional notation uses the probability expression* $\Pr(-)$ *which is applied to events. Hence it suggests that its argument* `even | atmost4` *is an event, and thus that* `|` *is an operation on events. This suggestion is misleading, since "*$\Pr(A|B)$ *is usually interpreted as "(probability of A) given B" rather than "probability of (A given B)," since "A given*

B" is not defined in logic." (quoted from[6]). Conditioning is an operation between states and predicates (events, if you like), and not between two events.

[6] G. Schay. An algebra of conditional events. *Journ. Math. Analysis and Appl.*, 24:334–344, 1968

Here is the mathematical account, see[7] for details. Let ω be a state and p a predicate, both with domain X. If the validity $\omega \models p$ is non-zero, one can form the conditional state $\omega|_p$, which is written in Python as ω/p. This new state is a revision of ω, given the evidence p. This conditional state $\omega|_p$ is defined by the formula:

[7] B. Jacobs. From probability monads to commutative effectuses. *Journ. of Logical and Algebraic Methods in Programming*, 156, 2017, to appear

$$\omega|_p = \sum_{x \in X} \frac{\omega(x) \cdot p(x)}{\omega \models p} |x\rangle. \tag{1.3}$$

Thus, $\omega|_p$ is a normalised product of ω and p, where normalisation happens via dividing by the validity $\omega \models p$.

Conditioning is a fundamental operation that forms the basis of (statistical) learning: updating one's knowledge, in the form of probabilities of domain elements, in the light of evidence, given by a predicate.

Conditioning satisfies the following basic equations.

$$\omega|_{\mathbf{1}} = \omega \qquad \text{and} \qquad \big(\omega|_p\big)|_q = \omega|_{p \& q} = \big(\omega|_q\big)|_p.$$

In addition there is Bayes' rule:

$$\omega|_p \models q \;=\; \frac{\omega \models p \,\&\, q}{\omega \models q}. \tag{1.4}$$

Further, for a predicates `p` on state `s` and `q` on `t` one can do conditioning component-wise via weakening:

`(s @ t) / (p @ truth(t.dom))`	is	`(s / p) @ t`
`(s @ t) / (truth(s.dom) @ q)`	is	`s @ (t / q)`

Here is an illustration.

```
>>> s = random_state(range(2))
>>> t = random_state(range(3))
>>> p = random_pred(range(2))
>>> (s @ t) / (p @ truth(t.dom))
0.131|0,0> + 0.161|0,1> + 0.266|0,2> + 0.104|1,0> + 0.127|1,1> + 0.21|1,2>
>>> (s / p) @ t
0.131|0,0> + 0.161|0,1> + 0.266|0,2> + 0.104|1,0> + 0.127|1,1> + 0.21|1,2>
```

Example 3 *There is a trick to turn an unfair coin into a fair, due to von Neumann. It is usually described as*[8]

[8] The three steps are as on `https://en.wikipedia.org/wiki/Fair_coin`

1. *Toss the coin twice.*
2. *If the results match, start over, forgetting both results.*

3. *If the results differ, use the first result, forgetting the second.*

Here we describe this idea not via repetition, but by discarding the matching results via conditioning. This works as follows.

```
>>> r = random.uniform(0,1)
>>> s = flip(r)
>>> s
0.539|True> + 0.461|False>
>>> s @ s
0.29|True,True> + 0.248|True,False> + 0.248|False,True> + 0.213|False,False>
```

We see that the 'middle' two cases, where the results of the two coins differ, have equal probability, given by $r \cdot (1 - r)$ where r is the bias. Hence if we concentrate on those two cases via conditioning we get a fair coin:

```
>>> (s @ s / (yes_pred @ no_pred + no_pred @ yes_pred)) % [1,0]
0.5|True> + 0.5|False>
```

Example 4 *The following example*[9] *is copied from the probabilistic programming language Church*[10] *and translated to the current context. The example involves a number of diseases:*

- `LC` = *lung-cancer, with a priori probability* 0.01*;*
- `TB` = *tuberculosis, with a priori* 0.005*;*
- `CO` = *cold, a priori* 0.2*;*
- `SF` = *stomach-flu, with* 0.1*;*
- `OT` = *other, also with* 0.1 *a priori probability.*

We incorporate these five a priori probabilities in a single product state:

```
>>> prior = flip(0.01) @ flip(0.005) @ flip(0.2) @ flip(0.1) @ flip(0.1)
```

The above five diseases are translated into five predicates on this state, each addressing the appropriate component of the above state, via weakening:

```
>>> up = Predicate([1,0], [True,False])
>>> W = truth([True,False])

>>> LC = up @ W @ W @ W @ W
>>> TB = W @ up @ W @ W @ W
>>> CO = W @ W @ up @ W @ W
>>> SF = W @ W @ W @ up @ W
>>> OT = W @ W @ W @ W @ up
```

By construction, the validity `prior >= LC` *is* `0.01`*, and similarly for the other diseases.*

Next, one considers the following four symptom predicates, depending diseases.

[9] See https://probmods.org/conditioning.html#example-causal-inference-in-medical-diagnosis for the original description.

[10] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008

A discussion about the 'logical' description that is used here, in contrast to a 'channel' description may be found later on, in Example 17.

```
>>> cough = 0.5 * CO | 0.3 * LC | 0.7 * TB |  0.01 * OT
>>> fever = 0.3 * CO | 0.5 * SF | 0.2 * TB | 0.01 * TB
>>> chest_pain = 0.4 * LC | 0.5 * TB | 0.01 * OT
>>> short_breath = 0.4 * LC | 0.5 * TB | 0.01 * OT
```

The scalar multiplications in these formulas serve as "noisy-logical functions to describe the dependence of symptoms on disease".

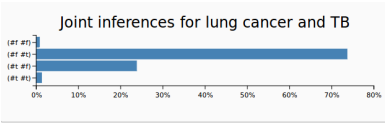
Suppose we observe all these sympotoms. Then we can revise the state to:

```
>>> post = prior / (cough & fever & chest_pain & short_breath)
```

The question that is asked (the 'query') in the original Church example is: what are the probabilities of the combinations lung cancer (LC) and tuberculosis (TC), given these observed symptoms? This is a typical conditional probability problem. The solution is obtained by taking the appropriate marginals of the conditional state:

```
>>> post % [1, 1, 0, 0, 0]
0.0161|True,True> + 0.242|True,False> + 0.741|False,True> + 0.000919|False,False>
```

The distribution can also be obtained with Curch at the website mentioned in sidenote 9. The resulting picture is included in the margin. Its True/False options correspond to the above ones, when read from below. Generating this distribution via Church with 100 samples takes about 20 seconds. It varies widely, as one can see by trying a few times at the website. In contrast, our answer pops up in a fraction of a second and is 'exact', in the sense that it does not depend on sampling but on the probabilistic computation.



Instead of using the above marginal distribution, we could also have asked for the validity of the two relevant attributes, as in:

```
>>> post >= LC & TB
0.0161267427839
```

This medical example illustrates how one builds and analyses models in the current setting:

- *a 'prior' state is defined that incorporates the prior probabilities;*
- *predicates on this state capture the evidence;*
- *observations translate into conditioning of the state;*
- *queries are answered either by inspecting the posterior states directly, or by asking for specific validities in those states.*

The following example illustrates another useful observation: conditioning can introduce entwinedness.

Example 5 *Consider the non-entwined joint state* `s` *below, the predicate* `p`*, and the conditional state* `t`*. This* `t` *is entwined, since it is not the product of its marginals:*

```
>>> s = flip(0.6) @ flip(0.8)
>>> s
0.48|True,True> + 0.12|True,False> + 0.32|False,True> + 0.08|False,False>
>>> p = Predicate([1, 1, 1, 0], s.dom)
>>> t = s / p
>>> t
0.522|True,True> + 0.13|True,False> + 0.348|False,True> + 0|False,False>
>>> (t % [1,0]) @ (t % [0,1])
0.567|True,True> + 0.0851|True,False> + 0.302|False,True> + 0.0454|False,False>
```

The fact that the state `t` *is entwined can also be deduced from* (1.1).

Our next example shows a fundamental property of entwined joint states: conditioning in one component influences the other component. This is the classical analogue of Einstein's *spukenhaften Fernwirkung* (spooky interaction at a distance), see Example 25. The same phenomenon occurs in Example 13 involving an entwined disease-and-mood state.

Example 6 *Here we show that conditioning in one part of an entwined state can influence the other part. Consider the joint state:*

```
>>> w = State([0.5, 0, 0, 0.5], [range(2), range(2)])
>>> w
0.5|0,0> + 0|0,1> + 0|1,0> + 0.5|1,1>
```

We consider the validity of the 'zero' point-predicate in the second component. As discussed before, we need to use weakening, via the truth predicate in the first component, to make the domains match.

```
>>> w >= truth(range(2)) @ point_pred(0, range(2))
0.5
```

If we condition in the first componet only, the validity of this predicate in the second component changes:

```
>>> w / (point_pred(0,range(2)) @ truth(range(2))) >= truth(range(2)) @ point_pred(0,range(2))
1.0
>>>  w / (point_pred(1,range(2)) @ truth(range(2))) >= truth(range(2)) @ point_pred(0,range(2))
0.0
```

If we condition in the first coordinate, and then project (marginalise) this coordinate away we obtain the point state corresponding to the point predicate:

```
>>> (w / (point_pred(0,range(2)) @ truth(range(2)))) % [0,1]
1|0> + 0|1>
>>> (w / (point_pred(1,range(2)) @ truth(range(2)))) % [0,1]
0|0> + 1|1>
```

Example 7 *This example illustrates the law of total probability in the current setting. We start from an arbitrary state and predicate and the associated validity:*

```
>>> s = random_state(range(4))
>>> p = random_pred(range(4))
>>> s >= p
0.326073622519
```

Next we wish to obtain this same probability via the law of total probability. This requires a test, that is, a set of predicates that add up to one. We choose a test of length 3. We create it in the following way, where the scalar `0.5` *is used to ensure that the scaled predicates* `0.5 * q1` *and* `0.5 * q2` *are orthogonal (summable). The predicate* `q3` *is then determined so that* `q1 + q2 + q3` *equals the truth predicate.*

```
>>> q1 = 0.5 * random_pred(range(4))
>>> q2 = 0.5 * random_pred(range(4))
>>> q3 = ~(q1 + q2)
>>> (s / q1 >= p) * (s >= q1) + (s / q2 >= p) * (s >= q2) +  (s / q3 >= p) * (s >= q3)
0.326073622519
```

This last formula is equivalent, by Bayes' rule (1.4)*, to:*

```
>>> (s >= q1 & p) + (s >= q2 & p) + (s >= q3 & p)
0.326073622519
```

In discrete and continuous probability such tests can be be factored out. But this is not the case in quantum probability, see at the end of Example 24.

1.3 Random variables

Random variables are used in probability theory to map elements of the sample space (domain) to real numbers. Random variables are closed under various operations like pointwise sum, product, scalar multiplication. Such operations on random variables are pre-defined in *EfProb*.

Random variable generalise predicates: where a predicate on a domain D is a function $D \to [0,1]$, a random variable is a function $D \to \mathbb{R}$. The notion of validity extends from predicates to random variable. Thus, besides validity `s >= p` for a predicate `p` in a state `s`, we can also have validity `s >= rv` of a random variable `rv` in a state `s`. This is the expected value. Random variables are define in the same way that predicates are defined.

In statistics random variables are crucial for expected values, variance and standard deviation, and also for covariance and correlation. This will be illustrated below.

1.3.1 *Expected value, variance, and standard deviation*

Given a state on the domain, one can then compute the expected value, or variance. Here is a simple example where this is useful.

> If it rains, an umbrella sales person can earn €100 per day. If it is a fair weather day (s)he can lose €20 per day. What is the expected return if the probability of rain is 0.3?

In an example like this it is important first to look for the state. Here model it as `flip(0.3)`, with domain `[True,False]`, describing whether or not it is raining. We would like to map the domain element `True` to `100` and the domain element `False` to `-20`. This is achieved in *EfProb* via:

```
>>> rain_state = flip(0.3)
>>> umbrella_sales_rv = RandVar([100, -20], [True,False])
>>> rain_state >= umbrella_sales_rv
16.0
>>> rain_state.expectation(umbrella_sales_rv)
16.0
```

We see that a random variable is defined by a list and a domain (also a list). The first list gives the real number values corresponding to the elements of the domain. A state has an expectation method `expectation`, which takes a random variable as argument. This state must have the same domain as the random variable. In the above example the validity of the random variable is printed first, giving the expected value (expectation). In *EfProb* there is also an `expectation` method on random variables that computes the expected value, taking a state as argument.

The above random variable can also be obtained by suitably multiplying yes/no point predicates with a scalar, as in:

```
>>> umbrella_sales_alt_rv = 100 * yes_pred + (-20) * no_pred
>>> rain_state.expectation(umbrella_sales_alt_rv)
16.0
```

In a similar way one obtains the variance or standard deviation via the methods `variance` and `st_deviation`, as illustrated below.

The average of a sequence of sample data, say `[2, 6, 4, 1, 10]` can be computed by turning the list of data into a random variable and computing the expected value, variance and standard deviation wrt. the uniform state:

```
>>> data = [2, 6, 4, 1, 10]
>>> dom = Dom(range(len(data)))
>>> s = uniform_state(dom)
>>> rv = RandVar(data, dom)
>>> s.expectation(rv)
```

```
4.6
>>> s.variance(rv)
10.24
>>> s.st_deviation(rv))
3.2
```

Here is another example of a random variable, now defined on (the domain of) the product state of a pair of dices. Instead of using a list, we use a function as first argument. It maps a pair $(x, y) \in \{1,2,3,4,5,6\} \times \{1,2,3,4,5,6\}$ to the sum $x + y$.

```
>>> fdice = uniform_state([1,2,3,4,5,6])
>>> twodice = fdice @ fdice
>>> sum_rv = RandVar.fromfun(lambda x,y: x+y, twodice.dom)
>>> twodice.expectation(sum_rv)
7.0
```

The *EfProb* framework also supports *conditional* expectation, simply via conditional states. For instance, the expected sum values of two dices given that both of them are even (or odd) are:

```
>>> even = Predicate([0,1,0,1,0,1], [1,2,3,4,5,6])
>>> odd = ~even
>>> (twodice / (even @ even)).expectation(sum_rv)
8.0
>>> (twodice / (odd @ odd)).expectation(sum_rv)
6.0
```

We can use Python as 'meta-language' to define for instance a function that returns the expected value of `n` dices:

```
>>> def sums_exp(n): return (fdice ** n).expectation(
...         RandVar.fromfun(lambda *xs: sum(xs), (fdice ** n).dom))
>>> sums_exp(1)
3.5
>>> sums_exp(2)
7.0
>>> sums_exp(3)
10.5
>>> sums_exp(8)
28.000000000186169
```

The latter computation takes in the order of 15 seconds on an ordinary laptop. The domain has size $6^8 = 1.679.616$ elements. The function involved uses a Python trick for arbitrary-length sequences of arguments, namely by adding a star in `lambda *xs: sum(xs)`.

One sees a pattern emerging, namely that `sums_exp(n)` equals `3.5 * n`. However, such formulas cannot be proven within *EfProb*. It can only be tested, for small numbers.

Below is a variation where we look at the 'conditional' expected value for `n` dices with `even` outcome, via conditional states.

```
>>> def even_sums_exp(n): return ((fdice ** n) / (even ** n)).expectation(
...         RandVar.fromfun(lambda *xs: sum(xs), (fdice ** n).dom))
>>> even_sums_exp(1)
```

Here the pattern is `even_sums_exp(n)` equals `4 * n`.

```
4.0
>>> even_sums_exp(2)
8.0
>>> even_sums_exp(8)
32.0
```

When the domain D of a state `s` is itself a subset $D \hookrightarrow \mathbb{R}$ of the real numbers $\mathbb{R}$, one can use a trivial inclusion function as random variable. In statistics, this trivial function is often omitted. This can also be done in *EfProb*, as shown in the next dice illustration, where, recall, the domain is given by the numbers $\{1,2,3,4,5,6\} \hookrightarrow \mathbb{R}$.

```
>>> fdice.expectation()
3.5
>>> fdice.variance()
2.91666666667
>>> fdice.st_deviation()
1.707825127659933
```

1.3.2 *Covariance and correlation*

For expectation (and variance and standard deviation) one needs a state together with a random variable on the same domain. For covariance and correlation one needs a state and *two* random variables on the same domain. Sometimes the state is omitted when it is trivial (uniform), and sometimes the random variables are omitted when they are trivial (projections). We shall show how this works in *EfProb* via illustrations.

We start with two samples, given by lists of numbers of the same length. We would like to learn their covariance and correlation. Implicitly one use a uniform state, with the common length as size.

```
>>> Xs = [5, 10, 15, 20, 25]
>>> Ys = [10,  8, 10, 15, 12]
>>> N = len(Xs)
>>> dom = Dom(range(N))
>>> s = uniform_state(dom)
>>> X_rv = RandVar(Xs, dom)
>>> Y_rv = RandVar(Ys, dom)
```

We can now compute not only expectations and variances, but also covariances and correlations. The latter take both random variables as argument:

```
>>> s.expectation(X_rv)
15.0
>>> s.expectation(Y_rv)
11.0
>>> s.variance(X_rv)
```

```
50.0
>>> s.variance(Y_rv)
5.6
>>> s.covariance(X_rv, Y_rv)
11.0
>>> s.correlation(X_rv, Y_rv)
0.657375735134
```

In the next example the covariance and correlation of a joint state are computed. Now the random variables are left implicit. This works when the product domain $D_1 * D_2$ consists of of subdomains $D_i \hookrightarrow \mathbb{R}$ of the real numbers.

```
>>> X = [1,2]
>>> Y = [1,2,3]
>>> w = State([1/4, 1/4, 0, 0, 1/4, 1/4], [X, Y])
>>> w
0.25|1,1> + 0.25|1,2> + 0|1,3> + 0|2,1> + 0.25|2,2> + 0.25|2,3>
>>> w.covariance()
0.25
>>> w.correlation()
0.707106781187
```

Like at the end of the previous subsection, no explicit random variables are given. They are inserted implicitly in *EfProb*, namely as projection random variables. They can also be added explicitly, as in:

```
>>> def proj1_rv(joint_dom): return randvar_fromfun(lambda *x: x[0], joint_dom)
>>> def proj2_rv(joint_dom): return randvar_fromfun(lambda *x: x[1], joint_dom)
>>> w.covariance(proj1_rv(w.dom), proj2_rv(w.dom))
0.25
>>> w.correlation(proj1_rv(w.dom), proj2_rv(w.dom))
0.707106781187
```

1.4 Channels

There are several ways to understand channels, in discrete probability, namely as:

- indexed collections of states;
- stochastic matrices;
- probabilistic transition systems;
- conditional probability tables, in Bayesian networks;
- indexed collections of states with the same domain;
- Kleisli maps for the distribution monad $\mathcal{D}$, between finite sets.

Channels form an abstraction notion of 'map' in a probabilistic setting, not only in discrete, but also in continuous and in quantum probability. They can be used for forward state transformation and backward predicate transformation. They can be composed both sequentially and parallelly, and form the basic ingredients of a probabilistic programming language.

Since channels are maps, they have a *domain* and a *codomain*. These (co)domains are precisely as for states and predicates. We often write `c : dom -> cod` to indicate that `c` is a channel with domain `dom` and codomain `cod`. These domain and codomain of a channel `c` can be obtained via attributes `c.dom` and `c.cod`.

We shall see several ways to define channels.

- Channels can be defined via (stochastic) matrix, provided with a domain and codomain; this is illustrated in Examples 9 and 14 below.
- There are also special 'conditional probability table' functions which are convenient to define channels associated with a Bayesian Network, see Examples 16 and 17.
- A channel can also be obtained from a list of states, all with the same domain. This is done via a static method `from_states`, as in:

```
>>> c = Channel.from_states([flip(0.2), flip(0.3), flip(0.5)])
>>> c
Channel of type: range(0, 3) --> [True, False]
>>> c.get_state(1)
0.3|True> + 0.7|False>
```

  We see that when printed, only the domain and codomain of a channel are shown. This channel maps an element $0 \leq i < 3$ in `range(0,3)` to the i-th state in the list, used as argument to the function `from_states`. These states can be extracted via the `get_state` method.

- Such a mapping from elements of a domain to states is a 'Kleisli map' for the distribution monad $\mathcal{D}$, in categorical terminology. Given such a mapping, one can form an associated channel. For instance, the previous example can also be obtained via the function `chan_fromklmap`.

```
>>> d = chan_fromklmap(lambda i: flip(0.2) if i == 0 else
...                    flip(0.3) if i == 1 else flip(0.5), range(3), [True,False])
```

1.4.1 State and predicate transformation

Let `c` be a channel, with domain `c.dom`. Each state `s` whose domain `s.dom` is the same as `c.dom` can be transformed into a state `c >> s` with

domain `c.cod`. We first describe how such state transformation works. Subsequently we discuss predicate transformation `c << p`, which works against the direction of the channel.

For the time being we shall define channels as in Example 9: a channel from a domain with n elements to a domain with m elements is given by an $n \times m$ stochastic matrix M, presented in linear form. The elements M_{ij} of the matrix, for $0 \leq i < n$ and $0 \leq j < m$, satisfy $\sum_j M_{ij} = 1$, for each i. Thus, if a state with a domain with n elements is given by probabilities s_i, with $\sum_i s_i = 1$, then state transformation can be described as $M \cdot s$, that is, as application of the matrix M to the state s. Explicitly,

$$(M \cdot s)_j = \sum_i M_{ij} \cdot s_i. \tag{1.5}$$

Predicate transformation works in the other direction. If a predicate `p` has a domain `p.dom` that is equal to the codomain `c.cod` of a channel `c`, then we can form the predicate `c << p` with domain `c.dom`. It may be understood as the weakest precondition of `p` along `c`. If the channel `c` has a matrix M as in the previous paragraph, and predicate `p` involves probabilities p_j, then the pulled back predicate `c << p` has probabilities:

$$(p^T \cdot M)_i = \sum_j p_j \cdot M_{ij}. \tag{1.6}$$

The following general result will be illustrated in Example 9 below. It shows how state- and predicate-transformation behave appropriately wrt. validity, as expressed by the following transformations validity equation.

$$c \gg s \models p \;=\; s \models c \ll p. \tag{1.7}$$

State and predicate transformation are extremely useful operations. This will be illustrated in a series of examples. The first example below describes in detail how this works, and is hopefully useful for readers who have not seen channels before.

Example 8 is copied from a lecture "Learning Bayesian Networks" of Peter Szolovits.

Example 8 *Imagine a company that produces two flavours of candy, namely cherry and lime. They are sold together in bags with the following labels and mixtures:*

h1 *100% cherry*

h2 *75% cherry, 25% lime*

h3 *50% cherry, 50% lime*

h4 *25% cherry, 75% lime*

h5 *100% lime.*

The prevalence of these five bags is $(0.1, 0.2, 0.4, 0.2, 0.1)$ *respectively.*

We will be interested in the probabilities of these bags, given certain candies. Before posing and answering such questions precisely, we first model the situation. It is important to recognise a prior distribution and a channel in the above data.

First we formalise the domain and prior distribution for the bags.

```
>>> candy_dom = Dom(['cherry', 'lime'])
>>> bag_dom = Dom(['h1', 'h2', 'h3', 'h4', 'h5'])
>>> prior = State([0.1, 0.2, 0.4, 0.2, 0.1], bag_dom)
>>> prior
0.1|h1> + 0.2|h2> + 0.4|h3> + 0.2|h4> + 0.1|h5>
```

The above listing h1 – h5 gives for each bag a probability distribution over 'cherry' and 'lime'. For instance, for bag h1 it is `flip(1, candy_dom)` *and for h2 it is* `flip(0.75, candy_dom)`*, etc. The above listing thus gives a bag-indexed collection of such states on* `candy_dom`*. Such a collection of states, all with the same domain, can be represented as a channel* `bag_dom --> candy_dom`*. We can formalise this by simply listing these states as input for the function* `chan_from_states` *that produces a channel, which we call* `chan`*.*

```
>>> chan = chan_from_states([flip(1, candy_dom),
...                          flip(0.75, candy_dom),
...                          flip(0.5, candy_dom),
...                          flip(0.25, candy_dom),
...                          flip(0.0, candy_dom)], bag_dom)
>>> chan
Channel of type: ['h1', 'h2', 'h3', 'h4', 'h5'] --> ['cherry', 'lime']
```

We now ask ourselves the first question: if we pick an arbitrary candy, from an arbitrary bag, what is the probability of getting 'cherry'?

We solve this by taking the sum of the probability of each bag multiplied with the probability of getting a cherry in each bag. This becomes:

$$0.1 \cdot 1 + 0.2 \cdot 0.75 + 0.4 \cdot 0.5 + 0.2 \cdot 0.25 + 0.1 \cdot 0 \;=\; 0.5.$$

This is precisely what is computed in state transformation:

```
>>> chan >> prior
0.5|cherry> + 0.5|lime>
```

Thus, state transformation `chan >> prior` *by the channel* `chan` *turns the state* `prior` *on* `bag_dom` *into a state on* `candy_dom`*. It does so by summing multiplied probabilities as in Equation* (1.5).

We can recover indexed states of the channels via state transformation applied to a point state, as below. Alternatively, we can use that channels are 'callable', and thus accept inputs from `bag_dom` *directly:*

```
>>> chan >> point_state('h2', bag_dom)
0.75|cherry> + 0.25|lime>
```

```
>>> chan('h2')
0.75|cherry> + 0.25|lime>
```

The probability distribution for candies starting from the uniform distribution on bags is computed as:

```
>>> chan >> uniform_state(bag_dom)
0.5|cherry> + 0.5|lime>
```

It is thus the same as for the prior distribution.

We now turn to predicate transformation, for the above channel `chan`*. It transforms predicates on the domain* `candy_dom` *to predicates on* `bag_dom`*. Predicate transformation thus works in the opposite direction of the channel itself. Here is an example, using the point predicate on* `candy_dom` *for 'lime'.*

```
>>> lime_pred = point_pred('lime', candy_dom)
>>> lime_pred
cherry: 0 | lime: 1
>>> chan << lime_pred
h1: 0 | h2: 0.25 | h3: 0.5 | h4: 0.75 | h5: 1
```

Predicate transformation works like a weakest precondition calculation: given that we see, a lime, what is the probability for each of the bags from `bag_dom`*? Notice that bag h1 is impossible, since it contains no lime candies.*

Similarly we can compute the likelihoods of the bags given that we saw a cherry.

```
>>> chan << ~lime_pred
h1: 1 | h2: 0.75 | h3: 0.5 | h4: 0.25 | h5: 0
```

More interestingly, if we are 50% sure that we have a cherry and 20% sure that we have a lime, then the likelihoods of the bags is: we saw a cherry.

```
>>> chan << 0.5 * ~lime_pred + 0.2 * lime_pred
h1: 0.5 | h2: 0.425 | h3: 0.35 | h4: 0.275 | h5: 0.2
```

These probabilities are calculated as a sum of products, according to formula (1.6)*.*

Things really start to become interesting if we involving conditioning of states. This brings us into the world of Bayesian reasoning. We aks ourselves, into what does our prior distribution change if we see a lime candy? This new state is obtained by updating the prior with the transformed lime predicate:

```
>>> prior / (chan << lime_pred)
0|h1> + 0.1|h2> + 0.4|h3> + 0.3|h4> + 0.2|h5>
```

We see that bag h1 is impossible, h2 has become less likely, the probability of h3 is unchanged, and the probabilities of h4 and h5 have increased. This make sense because h4 and h5 have more lime than cherry candies.

We can ask next what the likelihoods of cherry and lime are in this new state, if we pick one from an arbitrary bag. For convenience, we give the new updated state a name first.

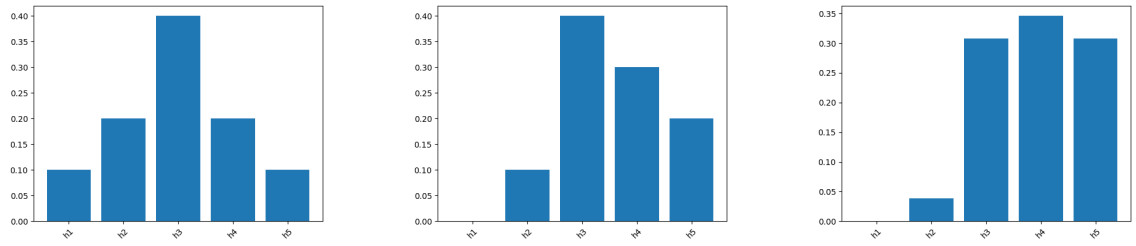
```
>>> s1 = prior / (chan << lime_pred)
>>> chan >> s1
0.35|cherry> + 0.65|lime>
```

Thus, after incorporating the evidence of a lime candy, a next lime is more likely than a cherry.

We check what the consequences of seeing another lime candy.

```
>>> s2 = s1 / (chan << lime_pred)
>>> s2
0|h1> + 0.0385|h2> + 0.308|h3> + 0.346|h4> + 0.308|h5>
>>> chan >> s2
0.269|cherry> + 0.731|lime>
```

Below we plot the prior bag probabilities, followed by the the updated probabilities after seeing one and two lime candies.



Example 9 *We continue with an illustration of the use of channels, which is typical for Bayesian reasoning, taken from*[11]*. For explanatary reasons we shall be very explicit about the domains involved. The setting is given by some disease with a priori probability of* 1%*. There is a test for the disease with the following 'sensitivity'. If someone has the disease, then the test is* 90% *positive; but if someone does not have the disease, there is still a* 5% *chance that the test is positive.*

[11] B. Jacobs and F. Zanasi. A predicate/state transformer semantics for Bayesian learning. In L. Birkedal, editor, *Math. Found. of Programming Semantics*, number 325 in Elect. Notes in Theor. Comp. Sci., pages 185–200. Elsevier, Amsterdam, 2016

```
>>> disease_domain = Dom(['D', '~D'])
>>> prior = flip(1/100, disease_domain)
>>> disease_pred = Predicate([1,0], disease_domain)
>>> prior
0.01|D> + 0.99|~D>
>>> prior >= disease_pred
0.01
>>> test_domain = Dom(['T', '~T'])
>>> test_pred = Predicate([1,0], test_domain)
>>> sensitivity = Channel([[9/10, 1/20], [1/10, 19/20]], disease_domain, test_domain)
>>> sensitivity.dom
```

```
['D', '~D']
>>> sensitivity.cod
['T', '~T']
```

The channel can be seen as a map `sensitivity : disease_domain -> test_domain`*. Here we use the description of the channel via a stochastic matrix. The same channel, in the style of Example 8, can be defined as:*

```
>>> sensitivity = chan_from_states([flip(9/10, test_domain),
...                                 flip(1/20, test_domain)], disease_domain)
```

We first use the channel to compute the probability that a test for an arbitrary person is positive. This is done via state transformation*, using the operator* `>>`*, as in:*

```
>>> sensitivity >> prior
0.0585|T> + 0.942|~T>
```

This probability `0.0585` *is the fraction* `117/2000`*. Notice that via state transformation* `sensitivity >> prior` *the state* `prior` *on the disease domain* `['D', '~D']` *is transformed into a state on the test domain* `['T', '~T']`*. State transformation* `>>` *happens via a 'push forward' in the direction of the channel.*

The a-priori-positive-test probability `117/2000` *can also be obtained via validity* `>=` *of predicates, in two different ways:*

```
>>> sensitivity >> prior >= disease_pred
0.0585
>>> prior >= sensitivity << test_pred
0.0585
```

First we ask for the validity of the disease predicate in the transformed state `sensitivity >> prior`*. Next we use* predicate transformation `<<` *to transform the test predicate on domain* `['T', '~T']` *into a predicate* `sensitivity << test_pred` *on the disease domain* `['D', '~D']`*. Thus, in predicate transformation a predicate is 'pulled back', in the opposite direction of the channel. The validity of this pulled-back predicate is then computed in the prior state. It yields the same outcome, via the general rule described in* (1.7)*.*

Once we have this pulled back predicate `sensitivity << test_pred` *we can use it to condition the prior state into a posterior state. A bit pedantically:*

```
>>> posterior = prior / (sensitivity << test_pred)
>>> posterior
0.154|D> + 0.846|~D>
```

This value `0.154` *is the truncated fraction* `18/117`*. It gives the probability of having the disease, given a positive test.*

Later on, in Example 16 we shall see more examples of Bayesian networks, also with *EfProb* support for defining the relevant channels from conditional probability tables (CPTs).

Example 10 *Capture and recapture is a methodology used in ecology to estimate the size of a population. So imagine we are looking at a pond and we wish to learn the number of fish. We catch twenty of them, mark them, and throw them back. Subsequently we catch another twenty, and find out that five of them are marked. What do we learn about the number of fish?*

The number of fish in the pond must be at least 20. Let's assume the maximal number is 300. We will be considering units of 10 fish. Hence the domain we are looking at, with the uniform 'prior' state, is:

```
>>> fish_domain = Dom([10 * i for i in range(2, 31)])
>>> fish_domain
[20, 30, 40, 50, 60, 70, 80, ..., 280, 290, 300]
>>> prior = uniform_state(fish_domain)
```

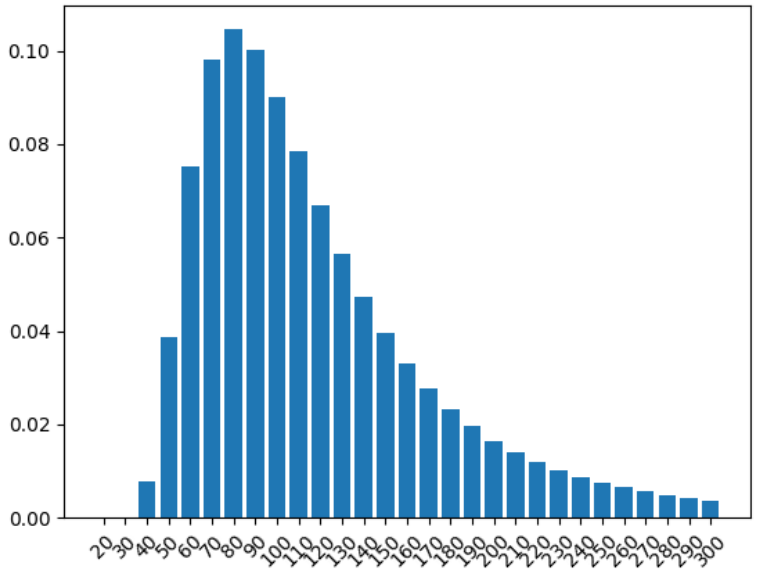
We now assume that 20 of the fish in the pond are marked. We can then compute for each value $20, 30, 40, \ldots$ *in the fish domain the probability of finding 5 marked fish when 20 of them are caught. In order not to complicate the calculations too much, we catch these 20 fish one by one, check if they are marked, and then throw them back. This means that the probability of catching a marked fish remains the same, and is described by a binomial distribution with parameters* $N = 20$ *and probability* $p = \frac{d}{20}$*, where* d *is a domain value. This is incorporated in the following channel, from the fish domain to the boolean domain.*

```
>>> N = 20
>>> chan = chan_fromklmap(lambda d: binomial(N, N/d), fish_domain, range(N+1))
>>> chan
Channel of type: [20, 30, 40, 50, 60, 70, 80, ..., 280, 290, 300] --> range(0, 21)
```

Once this is set up, the action can start. We construct a posterior state by updating the prior with the information that five marked fish have been found. The latter is expressed via a transformed predicate, as below.

```
>>> posterior = prior / (chan << point_pred(5, range(N+1)))
>>> posterior.expectation()
116.491929836
>>> posterior.plot()
```

This last line generates the diagram below, describing the likelihoods of the various numbers of fish in the pond.



Parameter learning is an important area in probability theory. We illustrate how to use channels in a structural way, via a simple example where the unkown bias of a coin is inferred from a series of consecutive 'head' and 'tail' observations. Here we learn the bias using discrete probabilitie distributions, which are iteratively updated with each observation. In the next chapter, in Example 21, we describe how this can be done using continuous distributions. Structurally, the approach remains the same.

Example 11 *In this example we look at the following classical question: suppose we are given a coin with an unknown bias, and we observe the following sequence of heads (1) and tails (0):*

$$0, 1, 1, 1, 0, 0, 1, 1$$

What can we say about the bias of the coin?

The approach is as follows. The unknown bias is a number in the unit interval $[0, 1]$*. We chop this interval up in* N *discrete parts and consider a probability distribution over these* N *parts. Initially, we know nothing about the bias of the coin, so our prior distribution will be a uniform one. We then update (revise) the state with each observation.*

In this example we choose as parameter $N = 20$ *and chop up the unit interval in the following way, using a precision of three decimals.*

```
>>> N = 20
>>> precision = 3
>>> bias_dom = Dom([floor((10 ** precision) * (i+1)/(N+1) + 0.5) / (10 ** precision)
```

```
...                 for i in range(N)])
>>> bias_dom
[0.048, 0.095, 0.143, 0.19, 0.238, 0.286, 0.333, 0.381, 0.429, 0.476,
   0.524, 0.571, 0.619, 0.667, 0.714, 0.762, 0.81, 0.857, 0.905, 0.952]
>>> prior = uniform_state(coin_bias)
```

Next we define a channel that maps each element `r` *of this bias domain to the state* `flip(r)`*.*

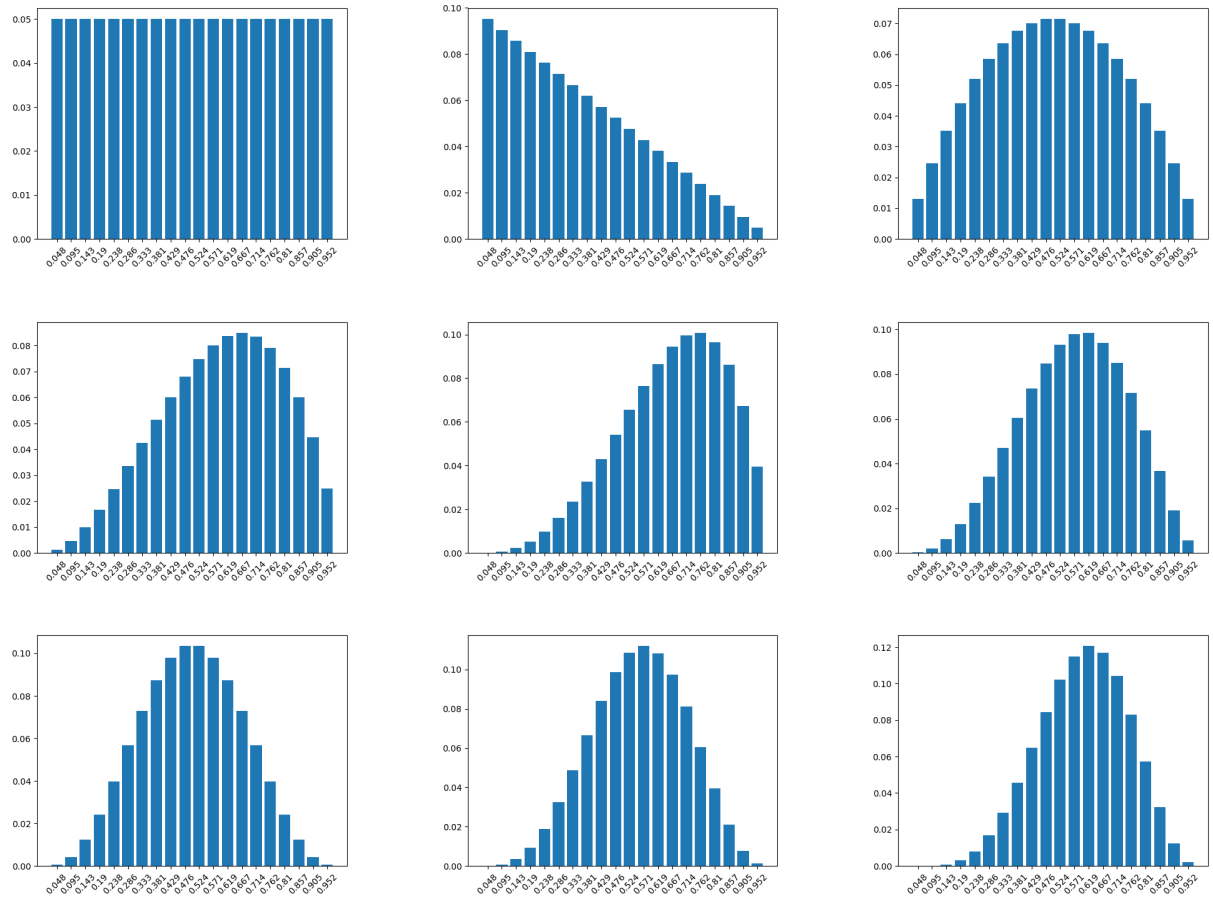
```
>>> chan = chan_fromklmap(lambda r: flip(r), bias_dom, bool_dom)
>>> chan >> prior
0.5|True> + 0.5|False>
```

Indeed, this prior uniform state corresponds to an unbiased coin.

Each head-or-tail observation induces an update of the state on `bias_dom` *with a yes-or-no predicate, translated via the channel. For convenience, we recall the definition of these yes/no predicates.*

```
>>> yes_pred = point_pred(True, bool_dom)
>>> no_pred = point_pred(False, bool_dom)
>>> observations = [0,1,1,1,0,0,1,1]
>>> s = prior
>>> s.plot()
>>> for ob in observations:
...     pred = yes_pred if ob==1 else no_pred
...     s = s / (chan << pred)
...     s.plot()
```

The nine states that are plotted in this code fragment look as follows.



These states approximate a (continuous) beta distribution.

In the end, after these 8 updates we can ask which coin corresponds to the bias distribution; it is obtained by state transformation.

```
>>> chan >> s
0.6|True> + 0.4|False>
```

Thus, after the observations $[0,1,1,1,0,0,1,1]$ *the probability of getting a* 1 *as next observation is* 0.6*.*

Alternatively we can ask for the expected value in this learned state — known as the 'posterior mean' — via the identity random variable:

```
>>> s.expectation(RandVar(bias_dom, bias_dom))
0.599974754805
```

Hence we arrive at a bias of o.6.

This final outcome is determined by the number of 0*'s and* 1*'s in the list of observations, and not by their order. This is because the order of conditioning does not matter in classical probability — unlike in quantum probability.*

The next example illustrates the 'denotation of a channel' construction, as used in the area of quantitative information flow.

Example 12 *It is not hard to see that a channel $c : dom \to [n]$ corresponds to an n-test of predicates on the domain dom. If we write $c(x) = c(x)(1)|1\rangle + \cdots + c(x)(n)|n\rangle$, then this n-test of predicates $p_1, \ldots, p_n$ is given by $p_i(x) = c(x)(i)$.*

In the EfProb library there is a function `predicates_from_channel` *that extracts this test as a list of predicates from a channel. Subsequently, it is used to define what is called the 'denotation of a channel', see e.g.* [12]. *The denotation produces a list of pairs (r_i, ω_i) of numbers $r_i \in [0, 1]$ adding up to one, and states ω_i. It is defined via validity and conditioning:*

[12] A. McIver, C. Morgan, G. Smith, B. Espinoza, and L. Meinicke. Abstract channels and their robust information-leakage ordering. In M. Abadi and S. Kremer, editors, *Princ. of Security and Trust*, number 8414 in Lect. Notes Comp. Sci., pages 83–102. Springer, Berlin, 2014

```
def channel_denotation(c, s):
    return [(s >= p, s/p) for p in predicates_from_channel(c)]
```

We illustrate this denotation function by copying Example 1 from[13].

[13] A. McIver, C. Morgan, G. Smith, B. Espinoza, and L. Meinicke. Abstract channels and their robust information-leakage ordering. In M. Abadi and S. Kremer, editors, *Princ. of Security and Trust*, number 8414 in Lect. Notes Comp. Sci., pages 83–102. Springer, Berlin, 2014

```
>>> X = Dom(['x1', 'x2', 'x2'])
>>> Y = Dom(['y1', 'y2', 'y3', 'y4'])
>>> c = Channel.from_states([State([1,   0,   0,   0  ], Y),
...                          State([0,   1/2, 1/4, 1/4], Y),
...                          State([1/2, 1/3, 1/6, 0  ], Y)], X)
>>> s = uniform_state(X)
>>> c >> s
0.5|y1> + 0.278|y2> + 0.139|y3> + 0.0833|y4>
>>> predicates_from_channel(c)
[x1: 1 | x2: 0 | x2: 0.5,
 x1: 0 | x2: 0.5 | x2: 0.333,
 x1: 0 | x2: 0.25 | x2: 0.167,
 x1: 0 | x2: 0.25 | x2: 0]
>>> cd = channel_denotation(c, s)
>>> cd
[(0.5, 0.667|x1> + 0|x2> + 0.333|x2>),
 (0.27777777777777779, 0|x1> + 0.6|x2> + 0.4|x2>),
 (0.1388888888888889, 0|x1> + 0.6|x2> + 0.4|x2>),
 (0.083333333333333329, 0|x1> + 1|x2> + 0|x2>)]
>>> convex_sum(cd)
0.333|x1> + 0.333|x2> + 0.333|x2>
```

We see how the four extracted predicates are obtained by 'vertically' reading out the probabilities of the three states in the definition of the channel. We have inserted spacing to increase the readability. The channel-denotation is printed subsequently as a list of four pairs (probability, state). Interestingly, the convex sum of these pairs is the original state `s`. *This is a general fact about the construction, see also*[14].

[14] B. Jacobs. Hyper normalisation and conditioning for discrete probability distributions. See arxiv.org/abs/1607.02790, 2016

1.4.2 *Sequential and parallel composition*

Channels can be composed both sequentially and in parallel. For sequential composition `d * c`, read as '`d` after `c`', it is required that the codomain of `c` is equal to the domain of `d`, as in:

$$\texttt{c.dom} \xrightarrow{\texttt{c}} \texttt{c.cod = d.dom} \xrightarrow{\texttt{d}} \texttt{d.cod}$$

The resulting channel `d * c : c.dom -> d.cod` is obtained by matrix multiplication. As a result, $*$ is an associative operation, with the identity channel (given by the identity matrix) as unit element.

Sequential composition of channels interacts with state- and predicate- transformation in the 'obvious' way: the left- and right hand side below are the same:

`(d * c) >> s`	is	`d >> (c >> s)`
`(d * c) << p`	is	`c << (d << p)`

Notice the reversal in order of the channels `c` and `d` in predicate transformation. The reason is that predicate transformation works in a backward direction.

For parallel composition `c @ d` of channels `c`, `d` there are no type restrictions. Thus we can form:

$$\texttt{c.dom + d.dom} \xrightarrow{\texttt{c @ d}} \texttt{c.cod + d.cod}$$

The sum `+` of domains involves a concatenation of lists of dimensions.

This parallel composition `@` for channels interacts appropriately with composition `@` for states and with `@` for predicates, in the sense that:

`(c @ d) >> (s @ t)`	is	`(c >> s) @ (d >> t)`
`(c @ d) << (p @ q)`	is	`(c << p) @ (d << q)`

The 'interchange law' for monoidal categories also holds:

`(c @ d) * (e @ f)`	is	`(c * e) @ (d * f)`

1.4.3 *Structural channels*

There are several channels which we call 'structural' because their main purpose is to provide structure to a network of channels. These structural channels include identity, discard (and projection), copy, swap, but also 'ancilla' and measure.

Identity channels For each domain `dom` there is an identity channel `idn(dom) : dom -> dom`. It is given by the identity matrix. For an

arbitrary channel `c : dom -> cod` the following three channels are the same.

```
idn(cod) * c        c        c * idn(dom)
```

This says that identity channels are units ('skip') for sequential composition $*$.

These identity channels seem rather useless, but they turn out to be handy in parallel compositions like `c @ idn(dom)`. Two identity channels in parallel forms itself an identity channel.

Discard channel For each domain `dom` there is discard channel `discard(dom) : dom -> []`. It throws away its input. Again this looks silly. But discard channels, in parallel with identity channels, give us projection channels. This is illustrated below, via the projections `proj1` and `proj2` which can do both marginalisation of states and weakening of predicates.

```
>>> s = State([1/12, 1/8, 1/4, 1/4, 1/6, 1/8], [bool_dom, range(3)])
>>> s
0.0833|True,0> + 0.125|True,1> + 0.25|True,2> + 0.25|False,0> + 0.167|False,1> + 0.125|False,2>
>>> s % [1,0]
0.458|True> + 0.542|False>
>>> s % [0,1]
0.333|0> + 0.292|1> + 0.375|2>
>>> proj1 = idn(bool_dom) @ discard(range(3))
>>> proj2 = discard(bool_dom) @ idn(range(3))
>>> proj1 >> s
0.458|True> + 0.542|False>
>>> proj2 >> s
0.333|0> + 0.292|1> + 0.375|2>
```

We continue the same example, now illustrating weakening of predicates via these same projection channels `proj1` and `proj2`.

```
>>> p1 = Predicate([1/4, 5/8], bool_dom)
>>> p2 = Predicate([1/2, 1, 1/12], range(3))
>>> p1 @ truth(range(3))
True,0: 0.25 | True,1: 0.25 | True,2: 0.25 | False,0: 0.625 | False,1: 0.625 | False,2: 0.625
>>> truth(bool_dom) @ p2
True,0: 0.5 | True,1: 1 | True,2: 0.0833 | False,0: 0.5 | False,1: 1 | False,2: 0.0833
>>> proj1 << p1
True,0: 0.25 | True,1: 0.25 | True,2: 0.25 | False,0: 0.625 | False,1: 0.625 | False,2: 0.625
>>> proj2 << p2
True,0: 0.5 | True,1: 1 | True,2: 0.0833 | False,0: 0.5 | False,1: 1 | False,2: 0.0833
```

Now that we know that marginalisation and weakening are intimately related via state/predicate transformation, we observe that we can also apply the transformations validity equation (1.7) and obtain the following informal equation:

$$\mathsf{marginalisation}(\omega) \models p \;=\; \omega \models \mathsf{weakening}(p).$$

This requires that marginalisation and weakening are done in the same coordinates. The following illustration continues the previous example.

```
>>> proj1 >> s >= p1
0.453125
>>> s >= proj1 << p1
0.453125
```

Copy and swap channels For each domain `dom` there is a copy channel `copy(dom) : dom -> dom @ dom`. It duplicates its input. Such copying exists in classical (discrete and continuous) probability, but not in quantum probability, because of the so-called no-cloning result. This copy channels is used for instance to define the 'graph' of a channel `c : dom -> cod` as a new channel `graph(c) : dom -> dom + cod`, in the following way.

```
def graph(c):
    return (idn(c.dom) @ c) * copy(c.dom)
```

This turns out to be a useful construction. There is also an n-fold copy `copy(dom, n)` which copies `n` times. The default value for `n` is `2`.

For each pair of domains `dom1` and `dom2` there is a swap channel:

```
swap(dom1,dom2) : dom1 @ dom2 -> dom2 @ dom1
```

It behaves in the obvious manner. These copy and swap channels play an important role in the channel that models a Bayesian network in Example 16.

We conclude this section with an example that illustrates the use of structural channels. At the same time it illustrates how entwinedness of states leads to 'crossover' influence, as in Example 6.

Example 13 *This examples introduces a variation on the disease-test illustration from Example 9. There we used a prior disease state:*

```
>>> disease_domain = Dom(['D', '~D'])
>>> prior = State([1/100, 99/100], disease_domain)
```

Now we add a second domain for mood, and use joint state for disease and mood, simply called `prior`.

```
>>> mood_domain = Dom(['M', '~M'])
>>> joint_prior = State([0.05, 0.5, 0.4, 0.05], [disease_domain, mood_domain])
>>> joint_prior
0.05|D,M> + 0.5|D,~M> + 0.4|~D,M> + 0.05|~D,~M>
```

This state describes a low probability for having the disease and a good mood, and also for not having the disease and a bad mood. The combination disease and bad mood, and no disease and good mood is more likely.

We recall from Example 9 that there is a test for the disease, described via a channel `sensitivity`*. We now ask ourselves the question: what are the mood probabilities before and after a positive test?*

The mood before the test is simply the second marginal of the joint state `joint_prior`*.*

```
>>> joint_prior % [0,1]
0.45|M> + 0.55|~M>
```

We first look at the probability of a positive test in the joint prior state. There are several equivalent ways to compute this. The first one takes the first marginal, applies the sensitivity state transformer and asks for the validity in the resulting state:

```
>>> sensitivity >> (joint_prior % [1,0]) >= test_pred
0.5175
```

One can also extend the sensitivity channel in parallel with the identity channel, apply it to the whole joint state, and look at the validity of the weakened predicate:

```
>>> (sensitivity @ idn(mood_domain)) >> joint_prior >= (test_pred @ truth(mood_domain))
0.5175
```

But, remembering the transformations validity equation (1.7), we note that we can get this same validity via predicate transformation:

```
>>> joint_prior >= (sensitivity @ idn(mood_domain)) << (test_pred @ truth(mood_domain))
0.5175
```

The latter can even be simplied to:

```
>>> joint_prior >= ((sensitivity << test_pred) @ truth(mood_domain))
0.5175
```

Recall that we are interested in the mood after a positive test. First we revise the prior with the information of a positive test, and then we compute the first and second marginal:

```
>>> joint_post = joint_prior / ((sensitivity << test_pred) @ truth(mood_domain))
>>> joint_post % [1,0]
0.957|D> + 0.0435|~D>
>>> joint_post % [0,1]
0.126|M> + 0.874|~M>
```

We thus see that, as a result of a positive test, the mood drops from a prior value of `0.45` *to a posterior value of* `0.126`*. In itself this is remarkable, since the sensitivity channel only acts on the first 'disease' part of the joint state, and not on the 'mood' part. The influence between these parts happens because 'disease' and 'mood' are entwined: the rise in disease probability leads to a lower mood.*

1.5 Hidden Markov models

We include two examples of hidden Markov models. Such models consist of a state space, with two operations (channels), one for observation and one for moving on to a next state. We consider two versions.

1. In the 'independent' version there are two separate channels obs: $X \to A$ and trs: $X \to X$ for observation and transition. They both act on the state space/domain X. The observation channel produces (distributions over) observable elements $a \in A$.
2. In the 'dependent' model there is a single channel ot: $X \to A \times X$, which combines both observation and transition.

Given an indepent model obs, trs we can form a dependent model, in the following way.

$$X \xrightarrow{\text{copy}} X \times X \xrightarrow{\text{obs}\times\text{trs}} A \times X$$

In the other direction, a dependent model gives rise to an independent one via projections:

$$\begin{array}{ccccc} & & & \xrightarrow{\pi_1} & A \\ X & \xrightarrow{\text{ot}} & A \times X & & \\ & & & \xrightarrow{\pi_2} & X \end{array}$$

Starting from an independent model, one can form a dependent one, and decompose it again; this yields the original model. But starting from a dependent model, one does not retrieve the original.

In this section we look at two example, borrowed from other languages, namely BLOG[15] and Church[16]. The first example is adapted from the reference manual of the probabilistic language BLOG[17].

Example 14 *We consider the four types of bases found in a DNA molecule: adenine (A), cytosine (C), guanine (G), and thymine (T). These four letters A, C, G, T will be used to form a domain, a prior distribution* `s0` *on that domain, and four predicates on* `s0` *for the different components.*

```
>>> ACGT = Dom(['A', 'C', 'G', 'T'])
>>> s0 = State([0.3, 0.2, 0.1, 0.4], ACGT)
>>> A = Predicate([1,0,0,0], ACGT)
>>> C = Predicate([0,1,0,0], ACGT)
>>> G = Predicate([0,0,1,0], ACGT)
>>> T = Predicate([0,0,0,1], ACGT)
>>> s0 >= A
0.3
```

[15] B. Milch, B. Marthi, S. Russell, D. Sontag, D. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In L. Getoor and B. Taskar, editors, *Statistical Relational Learning*. MIT Press, 2007

[16] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008

[17] B. Milch, B. Marthi, S. Russell, D. Sontag, D. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In L. Getoor and B. Taskar, editors, *Statistical Relational Learning*. MIT Press, 2007

The transition from one state to a next is given by a channel `trs`*. There is also a separate channel* `obs` *that gives the probability of each predicate in a particular state.*

```
>>> trs = Channel([[0.1, 0.3, 0.3, 0.3],
...                [0.3, 0.1, 0.3, 0.3],
...                [0.3, 0.3, 0.1, 0.3],
...                [0.3, 0.3, 0.3, 0.1]], ACGT, ACGT)
>>> obs = Channel([[0.85, 0.05, 0.05, 0.05],
...                [0.05, 0.85, 0.05, 0.05],
...                [0.05, 0.05, 0.85, 0.05],
...                [0.05, 0.05, 0.05, 0.85]], ACGT, ACGT)
```

The question that is addressed in the BLOG example is the following. Suppose we have consecutive observations: C, A, A, A, G; what is the resulting state/distribution?

The dynamics of hidden Markov models is given by (1) revise with observation, (2) make a transition. This is done consecutively by using both predicate transformation and state transformation:

```
>>> s1 = trs >> (s0 / (obs << C))
>>> s1
0.286|A> + 0.138|C> + 0.295|G> + 0.281|T>
>>> s2 = trs >> (s1 / (obs << A))
>>> s2
0.126|A> + 0.295|C> + 0.289|G> + 0.29|T>
>>> s3 = trs >> (s2 / (obs << A))
>>> s3
0.148|A> + 0.284|C> + 0.284|G> + 0.284|T>
>>> s4 = trs >> (s3 / (obs << A))
>>> s4
0.158|A> + 0.28|C> + 0.281|G> + 0.281|T>
>>> s5 = trs >> (s4 / (obs << G))
>>> s5
0.295|A> + 0.29|C> + 0.126|G> + 0.29|T>
```

Of course, one may handle such a sequence of observations automatically by iteration.

The above probabilities are the same as the ones in the BLOG reference manual. An important difference is that they are obtained here in an 'exact' manner, via actual computations, where in BLOG they are obtained via iterated sampling.

The next 'dependent' example use the Pólya urn model, a standard example in probability theory. It consists of an urn with an unkown mixture of black (B) and white (W) balls. When a ball is taken from the urn, its colour is inspected, and *two* balls of this same colour are added to the urn. Hence we see the combination of observation and state change that is typical for hidden Markov models.

Example 15 *We consider a Pólya urn with initially one black and one white ball in the urn and would like to know the distribution over all the possible observations after three step.*[18] *It is natural to use pairs* $(b,w) \in \mathbb{N}^2$ *of natural numbers as domain. Since we are interested in three steps only, we can limit the the domain to a finite set.*

[18] like in the problem description in the language Church https://probmods.org/chapters/05-observing-sequences.html

```
>>> N = 8
>>> num_dom = Dom(range(1,N)) @ Dom(range(1,N))
>>> prior = point_state((1,1), num_dom)
>>> col_dom = Dom(['B', 'W'])
```

We now define a 'dependent' model as a channel `num_dom --> col_dom @ num_dom`*.*

```
>>> c = chan_fromklmap(lambda b,w:  b/(b+w) * (point_state('B', col_dom)
...                                            @ point_state((b+1,w), num_dom))
...                     + w/(b+w) * (point_state('W', col_dom)
...                                  @ point_state((b,w+1), num_dom)),
...                     num_dom, col_dom @ num_dom)
```

Let the pair of numbers (b,w) *describe the contents of the urn. The probability of choosing a black ball is then* $\frac{b}{b+w}$*. With this probability, the next state is a product state consisting of colour* `B` *and successor state* $(b+1,w)$*. Similarly, with probability* $\frac{w}{b+w}$ *we move to the product state of* `W` *and* $(b,w+1)$*.*

Since the domain and codomain of the channel `c` *are different, we cannot simply compose* `c` *with itself. We have to take the types into accout, as in:*

$$\texttt{num\_dom} \xrightarrow{\texttt{c}} \texttt{col\_dom @ num\_dom} \xrightarrow{\texttt{idn(col\_dom) @ c}} \texttt{col\_dom @ col\_dom @ num\_dom}$$

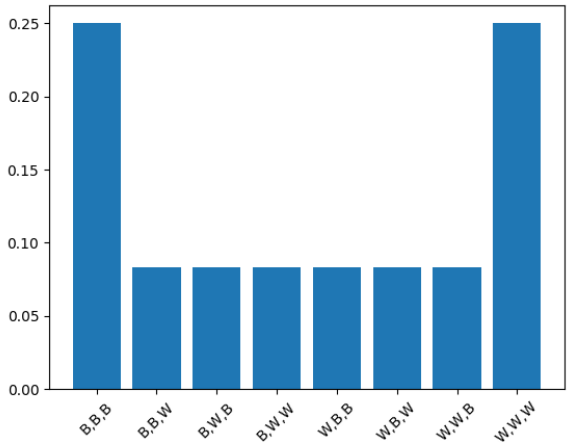
We do this another time, in a channel `d`*, and then discard the* `num_dom` *part:*

```
>>> d = (idn(col_dom @ col_dom) @ c) * (idn(col_dom) @ c) * c
>>> e = (idn(col_dom @ col_dom @ col_dom) @ discard(num_dom)) * d
>>> e
Channel of type: range(1, 8) * range(1, 8) --> ['B', 'W'] * ['B', 'W'] * ['B', 'W']
```

Hence now we are all set:

```
>>> prior = point_state((1,1), num_dom)
>>> e >> prior
0.25|B,B,B> + 0.0833|B,B,W> + 0.0833|B,W,B> + 0.0833|B,W,W> +
0.0833|W,B,B> + 0.0833|W,B,W> + 0.0833|W,W,B> + 0.25|W,W,W>
>>> (e >> prior).plot()
```

The latter line gives:



This same picture emerges in the Church setting.

1.6 Bayesian networks

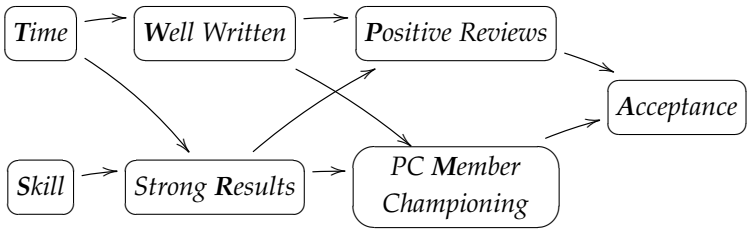
The *EfProb* library offers support for modeling Bayesian networks and for performing certain elementary computations within such networks. This includes functions to define prior states and conditional probability tables (cpt's), which are essentially channels.

A Bayesian network is given by a directed acyclic graph, consisting of nodes with arrows between them. These nodes can be understood as probability distributions on a two-element domain, whose elements are typically written as `t` for true and `f` for false. Nodes with no incoming arcs are called initial, and correspond to prior states. Arcs capture conditional probabilities. This will be explained via a Bayesian network taken from [19].

[19] B. Jacobs and F. Zanasi. A predicate/state transformer semantics for Bayesian learning. In L. Birkedal, editor, *Math. Found. of Programming Semantics*, number 325 in Elect. Notes in Theor. Comp. Sci., pages 185–200. Elsevier, Amsterdam, 2016

Example 16 *Consider the Bayesian network below that describes the*

various aspects of getting an academic paper accepted at a conference.



$\Pr(T)$	$\Pr(S)$
$\frac{4}{10}$	$\frac{7}{10}$

T	$\Pr(W)$
t	$\frac{8}{10}$
f	$\frac{3}{10}$

T	S	$\Pr(R)$
t	t	$\frac{9}{10}$
t	f	$\frac{6}{10}$
f	t	$\frac{4}{10}$
f	f	$\frac{1}{10}$

W	R	$\Pr(P)$
t	t	$\frac{8}{10}$
t	f	$\frac{5}{10}$
f	t	$\frac{6}{10}$
f	f	$\frac{1}{10}$

W	R	$\Pr(M)$
t	t	$\frac{4}{10}$
t	f	$\frac{1}{10}$
f	t	$\frac{3}{10}$
f	f	0

P	M	$\Pr(A)$
t	t	1
t	f	$\frac{7}{10}$
f	t	$\frac{8}{10}$
f	f	$\frac{1}{10}$

In EfProb this Bayesian network can be modeled as follows. There are the following pre-defined functions for states and predicates.

```
bnd = Dom(['t', 'f'])
def bn_prior(r): return State([r,1-r], bnd)
def bn_pred(r,s): return Predicate([r,s], bnd)
bn_pos_pred = bn_pred(1,0)
bn_neg_pred = bn_pred(0,1)
```

The abbreviation `bnd` *stands for 'Bayesian network domain'. It comes with two sharp predicates* `bn_pos_pred` *and* `bn_neg_pred` *that cover the true and false cases. The function* `bn_prior` *is used to define the initial/prior states. For the above Bayesian network example we put:*

```
>>> time = bn_prior(0.4)
>>> skill = bn_prior(0.7)
>>> prior = time @ skill
```

Next, the conditional probability tables are translated into channels via a function `cpt`*, as in:*

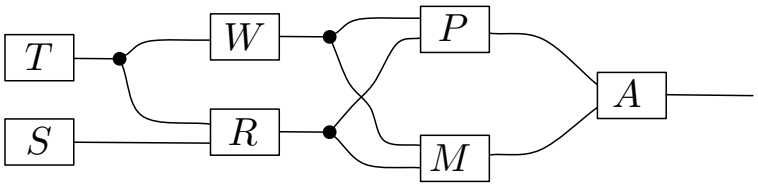
```
>>> well_written = cpt(0.8, 0.3)
>>> strong_results = cpt(0.9, 0.6, 0.4, 0.1)
>>> positive_reviews = cpt(0.8, 0.5, 0.6, 0.1)
>>> member_champion = cpt(0.4, 0.1, 0.3, 0.0)
>>> acceptance = cpt(1.0, 0.7, 0.8, 0.1)
```

These functions implicitly take care of the domains and codomains. Their codomain is always `bnd`*, but their domains depend on the number of probability parameters. For instance:*

```
>>> well_written.dom
['t', 'f']
>>> strong_results.dom
['t', 'f'] * ['t', 'f']
```

Indeed, the `well_written` *node has one incoming arcs, whereas the* `strong_results` *node has two.*

Before describing the whole network as a composition of channels we present it in slightly different form, namely as:



This picture clearly shows where we have to copy and where to swap wires. It helps to define the whole network, except the initial nodes, as a single channel:

```
>>> bn = acceptance \
...      * (positive_reviews @ member_champion) \
...      * (idn(bnd) @ swap(bnd,bnd) @ idn(bnd)) \
...      * (copy(bnd) @ copy(bnd)) \
...      * (well_written @ strong_results) \
...      * (copy(bnd) @ idn(bnd))
```

The probability that a paper gets accepted is now computed as:

```
>>> bn >> prior >= bn_pos_pred
0.47924
```

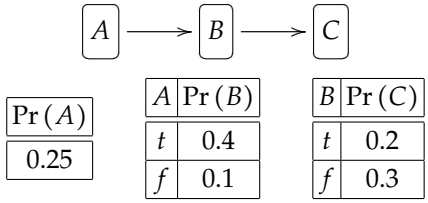
This same value is reported in[20]*, where it was computed in a more direct manner. There, a predicate* `E` *is used on the state* `prior` *with a additional weights for the various domain elements in* `prior`*. Given this predicate* `E`*, the probability of acceptance is different, and is computed in two (equivalent) ways.*

[20] B. Jacobs and F. Zanasi. A predicate/state transformer semantics for Bayesian learning. In L. Birkedal, editor, *Math. Found. of Programming Semantics*, number 325 in Elect. Notes in Theor. Comp. Sci., pages 185–200. Elsevier, Amsterdam, 2016

```
>>> E = Predicate([0.2, 0.4, 0.3, 0.6], prior.dom)
>>> bn >> prior/E
0.436|t> + 0.564|f>
>>> bn >> prior/E >= bn_pos_pred
0.435836923077
>>> prior/E >= bn << bn_pos_pred
0.435836923077
```

We proceed with another Bayesian network example with a different aim. It illustrates the difference between modeling a Bayesian network as a graph of channels, like in the previous example, and as set of logical formulas, like in Example 4.

Example 17 *Let's consider the following simple Bayesian network.*



Pr(A)
0.25

A	Pr(B)
t	0.4
f	0.1

B	Pr(C)
t	0.2
f	0.3

The state and channel representation of this network in EfProb is:

```
>>> prior = bn_prior(0.25)
>>> B_chan = cpt(0.4, 0.1)
>>> C_chan = cpt(0.2, 0.3)
```

The logical *represation of this network — as shown earlier in Example 4 in an illustration from the language Church[21] — uses the following formulas.*

[21] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008

```
>>> A_form = bn_pos_pred
>>> B_form = 0.4 * A_form | 0.1 * ~A_form
>>> C_form = 0.2 * B_form | 0.3 * ~B_form
```

The nodes of a Bayesian network are often interpreted as events. Let's follow this interpretation and look at various validities. First in the initial (prior) state the probability of 'event A' can be expressed in two different ways:

```
>>> prior >= bn_pos_pred
0.25
>>> prior >= A_form
0.25
```

Clearly, there is no difference. Next, how do we calculate the probability of 'event B'? We first describe the channel version and then the logical version:

```
>>> prior >= B_chan << bn_pos_pred
0.175
>>> prior >= B_form
0.175
```

Again there is no difference. We move on to 'event C'. We now get:

```
>>> prior >= B_chan << (C_chan << bn_pos_pred)
0.2825
>>> prior >= C_form
0.27485
```

Now there is a difference! Which approach is the right one?

The traditional way of calculating this probability via the 'total probabil-

ity' rules gives:

$$\begin{aligned}\Pr(B) &= \Pr(B \mid A) \cdot \Pr(A) + \Pr(B \mid \sim A) \cdot \Pr(\sim A) \\ &= 0.4 \cdot 0.25 + 0.1 \cdot 0.75 \\ &= 0.175 \\ \Pr(C) &= \Pr(C \mid B) \cdot \Pr(B) + \Pr(C \mid \sim B) \cdot \Pr(\sim B) \\ &= 0.2 \cdot 0.175 + 0.3 \cdot 0.825 \\ &= 0.285\end{aligned}$$

We see that the channel description — and not the logical description — coincides with the 'traditional' computation. The underlying reason is that the logical description only works for sharp predicates. That's why the differences show up only after two steps, and not after one step.

Still, all this remains a subtle matter, especially because our intuitions about logical connectives like 'and' and 'or' are Boolean-oriented and do not work well in a probabilistic setting with non-sharp (fuzzy) predicates. For instance, suppose, still in the setting of the above Bayesian network, that wish to know the probability of 'B and C'. According to Bayes' rule this is simply:

$$\Pr(B \wedge C) = \Pr(C \mid B) \cdot \Pr(B) = 0.2 \cdot 0.175 = 0.035.$$

In terms of channels there are two closely related formulations for 'B and C', namely:

```
>>> prior >= B_chan << (bn_pos_pred & C_chan << bn_pos_pred)
0.035
>>> prior >= (B_chan << bn_pos_pred) & (B_chan << (C_chan << bn_pos_pred))
0.04775
```

These outcomes are different because sequential conjunction `&` *is not preserved by predicate transformation* `<<`.

Finally, to re-emphasise that the 'logical' formulation is really not the right way to capture these probabilities we include:

```
>>> prior >= B_form & C_form
0.045905
```

The next example illustrates how 'evidence' is handled (propagated) through a Bayesian network, following[22]. This phenomenon is called *inference* or *belief update*.

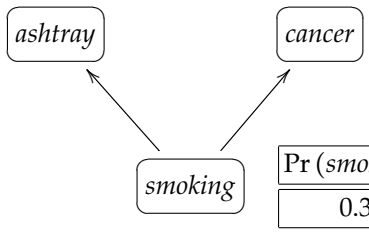
- **backward learning** corresponds to first (backward) predicate transformation, and then conditioning: $\omega|_{f^*(q)}$;
- **forward learning** corresponds to first conditioning and then (forward) state transformation: $f_*(\omega|_p)$.

[22] B. Jacobs and F. Zanasi. A predicate/state transformer semantics for Bayesian learning. In L. Birkedal, editor, *Math. Found. of Programming Semantics*, number 325 in Elect. Notes in Theor. Comp. Sci., pages 185–200. Elsevier, Amsterdam, 2016; and B. Jacobs and F. Zanasi. The logical essentials of Bayesian reasoning. See arxiv.org/abs/1804.01193, 2018

In *EfProb* notation the corresponding notation is `w / (f << q)` and `f >> (w / p)`.

Example 18 *We consider the relation between smoking and the presence of ash trays and (lung) cancer, in the following simple Bayesian network.*

smoking	Pr (*ashtray*)
t	0.95
f	0.25



smoking	Pr (*cancer*)
t	0.4
f	0.05

Pr (*smoking*)
0.3

Thus, 95% of people who smoke have an ashtray in their home, and 25% of the non-smokers too. On the right we see that in this situation a smoker has 40% chance of developing cancer, whereas a non-smoker only has 5% chance.

The question we want to address is: what is the influence of the presence or absence of an ashtray on the probability of developing cancer? Here the presence/absence of the ashtray is the 'evidence', whose influence is propagated through the network.

In the current setting of states, predicates and channels, the evidence is a predicate, which can be transformed along a channel via `<<`*, and used to update a state, which can then be transformed via* `>>`*. This will be described below. In addition, an alternative but equivalent approach will be described which first turns the above network into a joint state, and then uses 'crossover influence'*[23].

[23] Jacobs & Zanasi, forthcoming

In EfProb the network is represented via the definitions:

```
>>> smoking = bn_prior(0.3)
>>> ashtray = cpt(0.95, 0.25)
>>> cancer = cpt(0.4, 0.05)
```

We first consider the prior probabilities of smoking, ashtray, and cancer:

```
>>> smoking
0.3|t> + 0.7|f>
>>> ashtray >> smoking
0.46|t> + 0.54|f>
>>> cancer >> smoking
0.155|t> + 0.845|f>
```

The presence of an ashtray is given by the truth predicate `tt` *and is turned into predicate* `ashtray << tt` *on the state* `smoking`*. After updating this state, we transform it to an updated cancer probability. We can do this down-and-up propagation in one go:*

```
>>> cancer >> (smoking / (ashtray << tt))
0.267|t> + 0.733|f>
```

As expected, the presence of an ashtray gives a higher probability of cancer. Similarly, the absence of an ashtray gives a lower propbability:

```
>>> cancer >> (smoking / (ashtray << ff))
0.0597|t> + 0.94|f>
```

We also describe an alternative solution to this ashtray-cancer question. We first turn the above Bayesian network in a joint state, on the domain $\{t,f\} \times \{t,f\} \times \{t,f\}$, where the first component is for 'ashtray', the second one for 'smoking' and the third one for 'cancer'. We obtain this joint state via state tranformation using the 'tuple' channnel:

$$\{t,f\} \xrightarrow{\langle ashtray, idn, cancer \rangle} \{t,f\} \times \{t,f\} \times \{t,f\}$$

In EfProb this is done as follows, via first copying and then parallel composition `@`*.*

```
>>> joint = (ashtray @ idn(bnd) @ cancer) * copy(bnd,3) >> smoking
>>> joint
0.114|t,t,t> + 0.171|t,t,f> + 0.00875|t,f,t> + 0.166|t,f,f>
     + 0.006|f,t,t> + 0.009|f,t,f> + 0.0263|f,f,t> + 0.499|f,f,f>
```

(Recall that `bnd` *stands for 'Bayesian network domain', given by $\{t,f\}$.)*

This joint distribution contains the earlier-mentioned prior distributions as marginals:

```
>>> joint % [1,0,0]
0.46|t> + 0.54|f>
>>> joint % [0,1,0]
0.3|t> + 0.7|f>
>>> joint % [0,0,1]
0.155|t> + 0.845|f>
```

The presence of the ashtray evidence is now extended (weakened) to a predicate `tt @ truth(bnd) @ truth(bnd)` *on this joint state. We use it first for updating, and then we marginalise to obtain the third 'cancer' component that we are interested in:*

```
>>> (joint / (tt @ truth(bnd) @ truth(bnd))) % [0,0,1]
0.267|t> + 0.733|f>
```

The absence of an ashtray is handled via:

```
>>> (joint / (ff @ truth(bnd) @ truth(bnd))) % [0,0,1]
0.0597|t> + 0.94|f>
```

We see that the the joint-crossover approach yields the same cancer probability distributions as the earlier propagation approach, via predicate and state transformation.

2
Continuous Probability

Continous probability is standardly formulated in terms of probability measures on measurable spaces. In practical applications, these measurable spaces are given by a 'domain' subset $D \subseteq \mathbb{R}^n$ of the n-fold cartesian product $\mathbb{R}^n$ of the real numbers $\mathbb{R}$, for some number n. Moreover, the probability measures ω defined on (the σ-algebra associated with) the subset D are typically given a *probability density function* $f \colon D \to \mathbb{R}_{\geq 0}$, where $\mathbb{R}_{\geq 0} \subseteq \mathbb{R}$ is the subset of non-negative real numbers $r \in \mathbb{R}$ with $r \geq 0$. The resulting probability distribution is then given by $\omega(U) = \int_U f \, \mathrm{d}\mu$, where μ is the standard measure on the real numbers and U is a measurable subset $U \subseteq D$.

Given this situation, the *EfProb* library uses a restricted class of probability measures. Such a measure consists of:

1. a *domain* $D \subseteq \overline{\mathbb{R}}^n$, given by an n-dimensional rectangle of intervals in the extended real numbers $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$.

$$D = [\mathsf{lb}_1, \mathsf{ub}_1] \times [\mathsf{lb}_2, \mathsf{ub}_2] \times \cdots \times [\mathsf{lb}_n, \mathsf{ub}_n] \subseteq \overline{\mathbb{R}}^n$$

   where the $\mathsf{lb}_i, \mathsf{ub}_i \in \overline{\mathbb{R}}$ are the lower and upper bounds.

2. A probability density function $f \colon D \to \mathbb{R}_{\geq 0}$.

In the *EfProb* library there are no separate (Python) classes of discrete distributions and of continuous distributions. Instead, they are integrated in one class. This has the advantage that we can deal with discrete and continuous distributions simultaneously, for instance in a product state. It has the disadvantage the the structure of these distributions/states is a bit more complicated.

In the remainder of this chapter we will use 'state' as a synonym for 'distribution', either discrete or continuous. If we wish to emphasise one or the other, we will speak of a 'discrete state' or a 'continuous state'.

2.1 States

A continuous probability distribution (or: state) is given by a domain and a probability density function (pdf) from that domain to the positive real numbers. This will be illustrated in a series of examples.

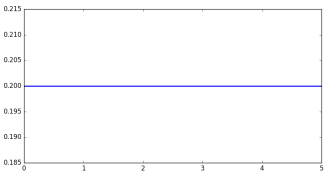
The most elementary continuous distribution is the continuous one, given on an interval, say $[0,5] \subseteq \overline{\mathbb{R}}$. It is defined as:

```
>>> u = uniform_state(R(0,5))
>>> u
State on R(0, 5)
```

In the discrete case we could just give a domain as a list of elements to the function `uniform_state`, see Section 1.1. But now, in the discrete case we use a special letter '`R`' to indicate that the domain is a subset of the real numbers: it is the interval $[0,5]$, given by its lower and upper bound in `R(0,5)`.

We also see in the above code snipped that printing the continous state `u` does not yield much information. We only learn that it is a `State on R(0,5)`. But we can *plot* the state, via:

```
>>> u.plot()
```



This produces the picture in the margin. It describes the probability density function (pdf) $f\colon[0,5] \to \mathbb{R}_{\geq 0}$ with constant value $f(x) = \frac{1}{5}$, such that the integral $\int_0^5 f(x)\,\mathrm{d}x$ equals 1. The latter property is essential for a probability distribution.[1]

One can restrict the domain that is plotting by passing an argument to the `plot` function, as in

```
>>> u.plot(R(1,4))
```

The probability density functions (pdf's) in states can be defined via Python's lambda notation, as in:

```
>>> s = State(lambda x: 0.5 * x, R(0,2))
```

Such a pdf must satisfy one crucial property: the integral over the whole domain must be equal to one. Predicates and validity will be describe systematically below, but here we already use them to perform this check, by testing the validity of the truth predicate on this domain.[2] For the state `s` that we just defined we get:

```
>>> s >= truth(R(0,2))
1.0
```

With user-defined states it is the responsability of the user that this condition is satisfied.

Domains may also be infinite, in one direction, or both, via the expression `inf`, as in:

[1] A pdf $f\colon D \to \mathbb{R}_{\geq 0}$ determines the probability of an event $U \subseteq D$ as the integral $\int_U f(x)\,\mathrm{d}x$, with outcome in the unit interval $[0,1]$. It is the size of the surface under f bounded by U. We shall describe this in probability in terms of validity later on.

[2] This validity test involves integration over the domain and could take some time. For that reason it is not automatically tested in *EfProb* upon state formation. Another reason, as discusssed in the introduction chapter is that such outcomes may be imprecise.

```
>>> u = State(lambda x: e ** -x, R(0, inf))
>>> u >= truth(u.dom)
1.0
>>> v = State(lambda x: e ** x, R(-inf, 0))
>>> v >= truth(v.dom)
1.0
>>> w = State(lambda x: 1/sqrt(pi) * e ** (-x*x), R)
>>> w >= truth(w.dom)
1.0
```

The latter state `w` is an example of a Gaussian distribution, see Subsection 2.1.1 below for more information.

One can also define states on product domains, via multiple lambda abstractions, see for instance:

```
>>> s = State(lambda x,y: 4 * x * y, [R(0,1), R(0,1)])
>>> s >= truth(s.dom)
1.0
>>> t = State(lambda x,y,z: 8/15 * x * y * z, [R(0,1), R(1,2), R(2,3)])
>>> t >= truth(t.dom)
1.0
```

The *EfProb* library contains functions for random states in discrete and quantum probability. Such random states do not exist in the continuous case. But there are several predefined standard distributions, which will be described below.

2.1.1 *Predefined distributions*

Several standard continous distributions are available in *EfProb*, such as Gaussian (normal), beta, gamma, and exponential distributions. They are based on their counterparts in Python's underlying `scypi` library.

A Gaussian (or: normal) distribution has a pdf g of the form:

$$g(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the 'mean' and σ is the standard deviation — so that σ^2 is the variance.

In *EfProb* this distribution is predefined as:

```
>>> g = gaussian_state(2,1)
>>> g
State on R
```

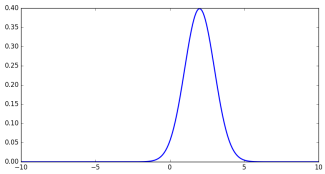
The first argument is the mean μ and the second one the standard deviation.

In *EfProb* it is possible to restrict the gaussian distribution to a bounded domain. This may be practical in certain situations. The

library then automatically adds a 'compensation' factor, so that the integral over the domain still equals one. This domain is added as optional third parameter, as in:

The default value for this domain parameter is R, for the whole real line.

```
>>> t = gaussian_state(2, 1, R(-10,10))
>>> t >= truth(t.dom)
1.0
>>> t.plot()
```

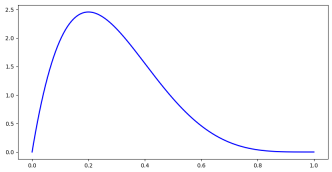


Such bounded gaussian states can be plotted, as shown in the margin.

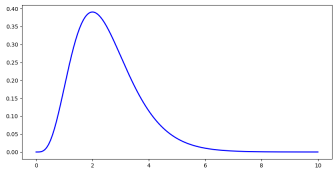
In a similar way there are for instance beta, gamma and exponential distributions:

```
>>> be = beta_state(2,5)
>>> be.plot()
>>> ga = gamma_state(5,2)
>>> ga.plot(R(0,10))
>>> ex = exponential_state(2)
>>> ex.plot(R(0,5))
```

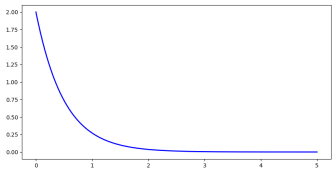
They lead to the following three plots.



beta



gamma



exponential

2.1.2 *Operations on states*

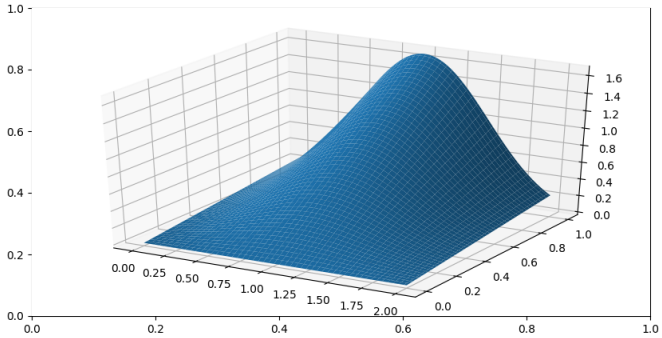
The operations that are described in Subsection 1.1.1 for discrete states also work for continuous states, via the same notation: product `@`, convex sum, and marginal `%`.

With the *EfProb* libarary one can form joint distributions via products `@`, also of discrete and continous probabilities.

```
>>> s = uniform_state(R(-1,1)) @ uniform_state(R(5,10))
>>> s
State on R(-1, 1) * R(5, 10)
>>> t = flip(0.3) @ s
>>> t
State on [True, False] * R(-1, 1) * R(5, 10)
```

We see that printing such states gives information about their domains. Only 1- and 2-dimensional continuous states with a bounded domain can be plotted. The plot below is produced via the following product of a Gaussian and Beta distribution.

```
>>> w = gaussian_state(1, 0.5, R(0,2)) @ beta_state(2,1)
>>> w.plot()
```



2.2 *Predicates and validity*

A continuous predicate on a domain D is a (measurable) function $p\colon D \to [0,1]$. Predicates can be defined just like states, as in:

```
>>> q = Predicate(lambda x: 1 if x < 0 else 0.5, R(-1,1))
```

Printing a predicate only shows its domain. But a predicate can be plotted, like a state — when its domain is 1-dimensional and bounded.

Continuous predicates in *EfProb* are closed under the same operations as discrete predicates: partial sum `+`, orthosupplement `~`, scalar multiplication $*$, sequential conjunction `&`, and parallel conjunction `@`. The binary operations `+` and `&` only work for predicates with the same domain, but `@` also works for predicates of different domains — even a mixture of discrete and continuous — and produces a predicate on the product domain.

Sequential conjunction `&` is commutative, like in discrete probability, but unlike in quantum probability. There is an associated sequential disjuntion operation `|`, defined via De Morgan rules.

In addition there there predicates `truth(dom)` and `falsity(dom)` which are constantly 1, resp. 0, on the whole domain. These truth and falsity functions work both for discrete and continuous domains `dom`.

Consider a predicate p and a state ω, with the same domain D. Thus p is a function $D \to [0,1]$, and ω is (identified with) a pdf $D \to \mathbb{R}_{\geq 0}$. The *validity* $\omega \models p$ in $[0,1]$ is in the continuous case defined as integral over the domain D of the product of the two functions::

$$\omega \models p \;=\; \int_D \omega(x) \cdot p(x)\, \mathrm{d}x. \tag{2.1}$$

This generalises the sum used in the discrete case (2.1). It gives the expected value of p in state ω.

In *EfProb/* Python the validity sign $\models$ is written as `>=`. It is computed via integration from the `scipy` library. For instance, for the above predicate `q` we have:

```
>>> u = uniform_state(R(-1,1))
>>> u >= q
0.75
>>> s = u @ flip(0.4)
>>> s >= ~q @ yes_pred
0.1
```

We can now see why we have checked the validities `s >= truth(s.dom)` several times in Section 2.1: they amount to the requirement that the integral $\int_D$ over the domain D of (the pdf of) `s` is one, see (2.1).

Example 19 *In*[3] *a situation is described with two fires on a ship that need to be contained within certain a time period in order to save the ship. The development of both fires is given by an exponential distribution, each with parameter 1.*

[3] S. Michels, A. Hommersom, P. Lucas, and M. Velikova. A new probabilistic constraint logic programming language based on a generalised distribution semantics. *Artificial Intelligence*, 228:1–44, 2015

```
>>> fires = exponential_state(1) @ exponential_state(1)
```

This is a joint state on the domain `[R(0,inf), R(0,inf)]`.

A rather artifical constraint is given in loc. cit. about the times within which the fires must be extinguished, in order to save the ship. We translate it literally into a predicate in EfProb, and calculate its validity:

```
>>> saved = Predicate(lambda t1, t2: 1 if t1 < 1.25 and t1 + 0.25 * t2 < 1.375
...                   else 0, [R(0,inf), R(0,inf)])
>>> fires >= saved
0.656931730836
```

This corresponds to the approximate probability 0.66 *given in the original source. Computing this probability involves a complicated integral, which takes about 25 sec. on a standard PC.*

The above predicate `saved` *contains some inner logical structure which can be made explicit in EfProb. We then get the same outcome:*

```
>>> p1 = Predicate(lambda t: t < 1.25, R(0,inf))
>>> p2 = Predicate(lambda t1, t2: 1 if t1 + 0.25 * t2 < 1.375 else 0,
...                [R(0,inf), R(0,inf)])
>>> fires >= (p1 @ truth(R(0,inf))) & p2
0.656931730836
```

2.2.1 *Conditioning*

We recall that conditioning involves updating (revising) a state ω in the light of certain evidence in the form of a predicate p. The result

is a conditional state, 'ω given p', written as $\omega|_p$. This new state is defined only when ω and p have the same domain D and have non-zero validity $\omega \models p$. The resulting probability density function $\omega|_p \colon D \to \mathbb{R}_{\geq 0}$ is defined as:

$$\omega|_p(x) = \frac{\omega(x) \cdot p(x)}{\omega \models p}. \tag{2.2}$$

Thus, $\omega|_p$ is a normalised product of ω and p. The normalisation ensures that the integral $\int_D \omega|_p(x)\,\mathrm{d}x$ equals one.

Bayes' rule holds in the continuous case in precisely the form (1.4) described for discrete probability.

In *EfProb/* Python the update of a state `s` with predicate `p` is written as `s / p`. The following archeological example is taken from[4,5].

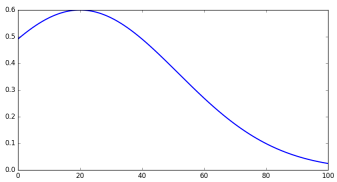
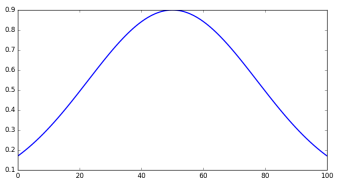
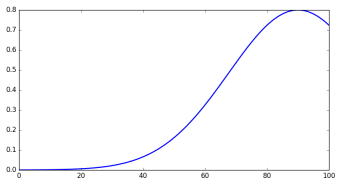
[4] B. Jacobs, B. Westerbaan, and A. Westerbaan. States of convex sets. In A. Pitts, editor, *Foundations of Software Science and Computation Structures*, number 9034 in Lect. Notes Comp. Sci., pages 87–101. Springer, Berlin, 2015

[5] B. Jacobs and F. Zanasi. A predicate/state transformer semantics for Bayesian learning. In L. Birkedal, editor, *Math. Found. of Programming Semantics*, number 325 in Elect. Notes in Theor. Comp. Sci., pages 185–200. Elsevier, Amsterdam, 2016

Example 20 *Imagine a tomb at an archeological site of which we already know that it is from the interval 0 – 100 AD. We like to learn its year of origin more precisely. This will be done by searching at the tomb site for three kinds of objects, labelled 1, 2, 3, of which we know the time of use more precisely. They are used to infer the age of the tomb. This knowledge is represented by three predicates given by the formulas:*

```
>>> dom = R(0,100)
>>> p1 = Predicate(lambda x: 0.6 * e ** (-1/2000 * (x - 20) ** 2), dom)
>>> p2 = Predicate(lambda x: 0.9 * e ** (-1/1500 * (x - 50) ** 2), dom)
>>> p3 = Predicate(lambda x: 0.8 * e ** (-1/1000 * (x - 90) ** 2), dom)
```

These three predicates look as follows.

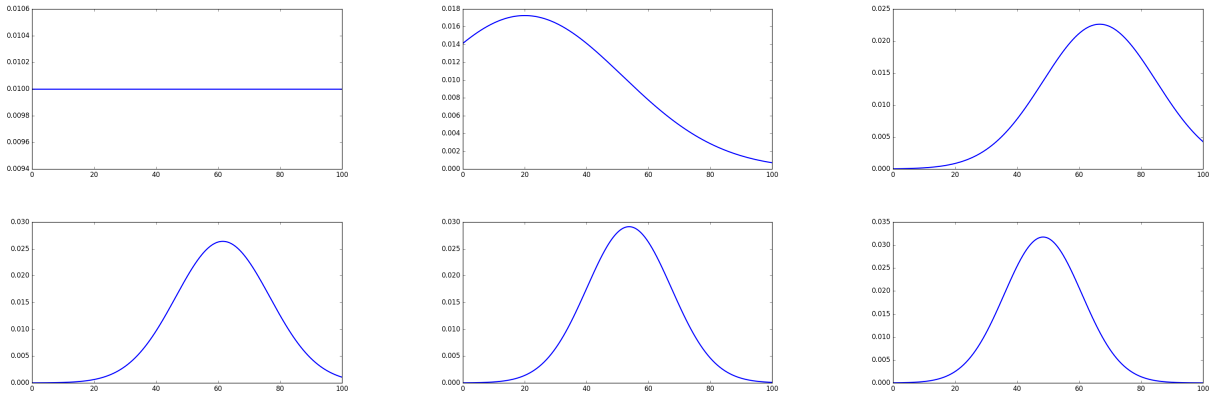
Initially each age of the tomb within the interval $[0,100]$ *is equally likely. Hence our prior is a uniform state* `u`. *This state is updated after each finding of object* $i \in \{1,2,3\}$. *For instance, after discovering an object of type 1, we change our state to* `u / p1`. *If we next find an object of type 3, our state becomes* `u / p1 / p3`, *etc.*[6]

We do this more systematically for a list of 5 successive objects.

[6] The state `u / p1 / p3` is the same as `u / p1 & p3`, and as `u / p3 & p1`, and as `u / p3 / p1`.

```
>>> findings = [1, 3, 2, 1, 1]
>>> u = uniform_state(dom)
>>> def pred(f): return p1 if f==1 else p2 if f==2 else p3
>>> for f in findings: u = u / pred(f)
```

This leads to the following successive states.



As will be described more systematically in the next section, the expected age value in the final state can be computed as:

```
>>> u.expectation()
48.4229427188
```

2.3 Random variables

Where continuous predicates on a domain D are (measurable) functions $p: D \to [0,1]$, random variable are more general functions $D \to \mathbb{R}$. These random variables are closed under more operations than predicates, in particular unrestricted addition and scalar multiplication.

Random variables are defined like predicates, via a function and a domain. The validity of a random variable in a state is defined in basically the same way as the validity of a predicate.

```
>>> s = uniform_state(R(0,5))
>>> rv = RandVar(lambda x: x, R(0,5))
>>> s >= rv
2.5
```

Random variable play an important role in the concept of expected value (expectation), variance, standard deviation, and also of covariance and correlaion. This works precisely as in the discrete case, and will be illustrated here via a few examples.

Expectation, variance *etc.* are methods on states, taking a random variable as argument. When the domain D of the state is subspace of the real numbers, then the random variable can be ommitted; the inclusion $D \hookrightarrow \mathbb{R}$ is then used implicitly. When we take a Gaussian state, the 'μ' and 'σ' arguments reappear of course as via the expectation and standard deviation methods:

```
>>> g = gaussian_state(-1.4, 2.9)
```

```
>>> g.expectation()
-1.4
>>> g.st_deviation()
2.9
```

One can compute that the expected value for the next state is $\frac{5}{9}$ and that its variance is $\frac{55}{1134}$.

```
>>> t = State(lambda x: 10/3 * x - 10/3 * x ** 4, R(0,1))
>>> t >= truth(R(0,1))
>>> t.expectation()
0.555555555556
>>> t.variance()
0.0485008818342
```

We turn to covariance of a joint state, where the domains are subsets of the real numbers. Again, the random variables are provided implicitly, as projections.

```
>>> w = State(lambda x,y: 4*x*y, [R(0,1), R(0,1)])
>>> w >= truth(R(0,1)) @ truth(R(0,1))
1.0
>>> w.covariance()
2.68546700673e-19
>>> w.correlation()
4.83384061211e-18
```

These outcomes are very small, through rounding errors. Mathematically they should be 0, since the state `w` is a product state, namely of `State(lambda x: 2*x, R(0,1))` with itself.

2.4 Channels

2.4.1 State and predicate transformation

The following parameter learning illustration is the continuous analogue of Example 11. The approach is basically the same: we only have to change the domain from discrete to continuous.

Example 21 *In this example we like to learn the bias of a coin from a sequence of heads and tails. The bias is a number from the unit interval* $[0,1]$. *In the discrete approach we chopped this unit interval up into intervals, and considered a discrete distribution over these intervals. In this chapter we work directly with a continuous distribution on the unit interval, represented in EfProb as* `R(0,1)`.

```
>>> bias_dom = R(0,1)
>>> prior = uniform_state(coin_bias)
```

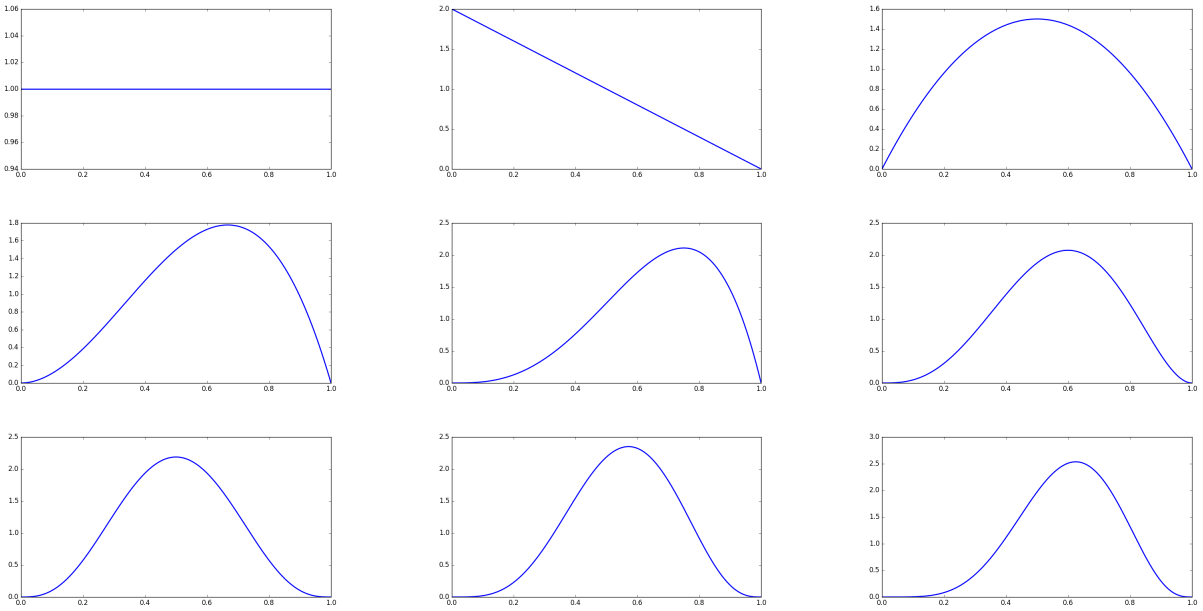
The channel definition is exactly as in the discrete case — except that the `bias_dom` *has a different meaning.*

```
>>> chan = chan_fromklmap(lambda r: flip(r), bias_dom, bool_dom)
>>> chan >> prior
0.5|True> + 0.5|False>
```

We use the same head (1) and coin (0) observations as in the discrete case. This allows us to compare the resulting pictures.

```
>>> observations = [0,1,1,1,0,0,1,1]
>>> s = prior
>>> s.plot()
>>> for ob in observations:
...     pred = yes_pred if ob==1 else no_pred
...     s = s / (chan << pred)
...     s.plot()
```

This leads to the following sequence of pictures.



These nine plots have the same shape as the (unnormalised) beta function $x \mapsto x^{\alpha-1} \cdot (1-x)^{\beta-1}$ *on* $[0,1]$ *for consecutively* (α,β)*-values* $(1,1)$, $(1,2)$, $(2,2)$, $(3,2)$, $(4,2)$, $(4,3)$, $(4,4)$, $(5,4)$, $(6,4)$. *The reason is that the beta function is the 'conjugate prior' of the flip distribution, see e.g.* [7]. *The big advantage of such conjugate priors is that they allow us to do updates via the 'hyper' parameters* α, β*, instead of via the distribution itself, as in the updates* `s / (chan << pred)` *in the above code fragment.*

We can now calculate for the final state `s` *after the above loop the best bias of the coin:*

```
>>> chan >> s
0.6|True> + 0.4|False>
```

[7] B. Jacobs. A channel-based perspective on conjugate priors. See arxiv.org/abs/1707.00269, 2017a

```
>>> s.expectation()
0.6
```

Hence we arrive at a bias of 0.6, precisely as in the discrete case in Example 11.

We include a similar parameter learning example, in a different context.

Example 22 *Imagine a company that tries out two different advertisement campaigns, labeled A and B, for the same product. Both campaigns are tried out on $N = 50$ individuals, with different success scores: 18 people respond by buying the product for campaign A, and 27 people for campaign B.*

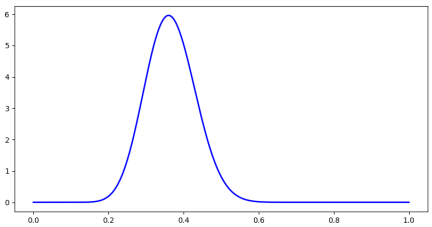
First we would like to learn the success rates for the two campaigns. These success rates are numbers $r \in [0,1]$ which are used to calculate the binomial probability of k successes out of N. This is done via the following channel from $[0,1]$ to $\{0,1,\ldots,N\}$.

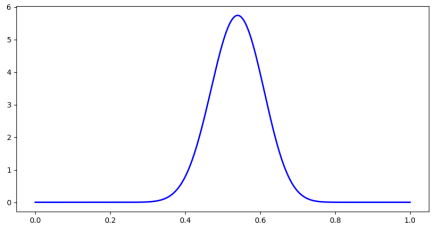
```
>>> N = 50
>>> c = chan_fromklmap(lambda r: binomial(N, r), R(0,1), range(N+1))
```

Following standard Bayesian parameter learning, we start from a uniform prior, and update with the respective (transformed) score predicates:

```
>>> prior = uniform_state(R(0,1))
>>> scoreA = 18
>>> scoreB = 27
>>> postA = prior / (c << point_pred(scoreA, range(N+1)))
>>> postB = prior / (c << point_pred(scoreB, range(N+1)))
>>> postA.plot()
>>> postB.plot()
```

The resulting plots for campaigns A and B are given below on the left and right.





It is clear that campaign B has a higher success rate. But now we wish to take the costs of the two campaigns into account. Each acquired customer yields a profit of 100, Euro say. The advertisement cost per participant is 10 for campaign A and 30 for campaign B. Which campaign is more cost-effective?

We compute this via two random variables on $\{0,1,\ldots,N\}$ that take these profits and costs into account:

```
>>> profitA = randvar_fromfun(lambda x: x * 100 - N * 10, range(N+1))
>>> profitB = randvar_fromfun(lambda x: x * 100 - N * 30, range(N+1))
>>> (c >> postA).expectation(profitA)
1326.92307692
>>> (c >> postB).expectation(profitB)
1192.30769231
```

We see that the net profit of campaign A is expected to be higher than for campaign B. Of course, we could have seen this difference by computing `scoreA*100-N*10` *and* `scoreB*100-N*30`*; this yields 1300 and 1200, respectively.*

As an aside: the latter two expected values can also be computed as validities of transformed random variables:

```
>>> postA >= (c << profitA)
1326.92307692
>>> postB >= (c << profitB)
1192.30769231
```

3
Quantum Probability

The reader is assumed to be familiar with the basics of quantum theory, see for instance[1].

The code fragments shown in this chapter require the `efprob_qu` library. It should be imported either on the command line at the beginning of a session, or in the beginning of a Python file.

This chapter follows the same pattern as the previous two, discussing states and predicates, random variables and channels, together with many illustrations and some basic explanations of the underlying mathematical structures. The quantum part of *EfProb* is separate from the (combined) discrete and continuous part. But discrete probability does exist as special case within the *EfProb* quantum world, as the 'diagonal' case in density matrices, see below. The operations on states and predicates and channels are in general the same in quantum probability as in the 'classical' non-quantum probability, with as notable exceptions:

- non-commutativity of certain operations, like sequential conjunction & and conditioning of states;
- non-copyability of states, because of the no-cloning property.

[1] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000; E. Rieffel and W. Polak. *Quantum Computing. A Gentle Introduction*. MIT Press, Cambridge, MA, 2011; and N.S. Yanovsky and M.A. Manucci. *Quantum Computing for Computer Scientists*. Cambridge Univ. Press, 2008

To do: comparison with other quantum languages like Quipper, Qwire, LIQ*Ui*|⟩, and also to QuTiP.

A. Green., P. LeFanu Lumsdaine, N. Ross, P. Selinger, and B. Valiron. Quipper: A scalable quantum programming language. In *Proc. 34th ACM SIGPLAN Conf. on Progr. Language Design and Implementation*, pages 333–342. ACM, 2013; J. Paykin, R. Rand, and S. Zdancewic. QWIRE: A core quantum circuit language. POPL, 2017; and D. Wecker and K. Svore. Liq*ui*| >: A software design architecture and domain-specific language for quantum computing, 2014. See arxiv.org/abs/1402.4467

3.1 States

States in quantum theory are very different from states in discrete probability and also from states in continuous probability. Nevertheless, they do include discrete probabilistic states as special case, as we shall see. Quantum states — or simply 'states' in this chapter — are represented as *density matrices*. A state of dimension n is a represented as a matrix ρ of dimension $n \times n$, which is positive and has trace one: the elements on the diagonal add up to one. We sometimes use the standard notation M_n for the set of arbitrary $n \times n$ (square) matrices over the complex numbers $\mathbb{C}$. Thus, a state of dimension n is a special element $\rho \in \mathsf{M}_n$, given by $n \times n$ complex numbers. Since it

is positive with trace 1, all its elements on the diagonal are real numbers, that add up to one. Here we only consider finite-dimensional states.

Let's see some examples, starting with dimension 2. States of this dimension are called *qubits*. The two most familar ones are written as `ket(0)`, `ket(1)`, `plus`, `minus` in:

```
>>> ket(0)
[[ 1.  0.]
 [ 0.  0.]]
>>> ket(1)
[[ 0.  0.]
 [ 0.  1.]]
>>> plus
[[ 0.5+0.j  0.5+0.j]
 [ 0.5+0.j  0.5+0.j]]
>>> minus
[[ 0.5+0.j -0.5+0.j]
 [-0.5+0.j  0.5+0.j]]
```

The notation `0.j` is part of python's notation for the imaginary part of complex numbers. In all of the above examples the imaginary part is zero.

The matrix of a state `s` can be accessed as `s.array`. It is what is (pretty) printed if you print a state.

Please note that we deviate to some extent from standard quantum notation where the ket notation $|0\rangle$ is used for the unit vector $\binom{1}{0} \in \mathbb{C}^2$, and $|1\rangle$ for $\binom{0}{1} \in \mathbb{C}^2$. Here `ket(0)` means the corresponding density matrix, which is conventionally written as $|0\rangle\langle 0|$. Similarly, `ket(1)` is $|1\rangle\langle 1|$. The two other states above, `plus` and `minus`, are, respectively, the density matrices:

$$|+\rangle\langle +| \quad \text{and} \quad |-\rangle\langle -| \quad \text{where} \quad \begin{cases} |+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \end{cases}$$

States can be defined 'manually' as objects of the class `State`. The constructor of this class takes an array as argument, together with a domain. For instance, the states `ket(0)` and `ket(1)` could be defined as:

```
ket(0) = State(np.array([1,0],
                        [0,0]]), [2])
ket(1) = State(np.array([0,0],
                        [0,1]]), [2])
```

The Bell states are pre-defined via outer products as:

```
bell00_vect = np.array([1,0,0,1])
bell01_vect = np.array([0,1,1,0])
bell10_vect = np.array([1,0,0,-1])
bell11_vect = np.array([0,1,-1,0])
```

```
bell00 = State(0.5 * np.outer(bell00_vect, bell00_vect), [2,2])
bell01 = State(0.5 * np.outer(bell01_vect, bell01_vect), [2,2])
bell10 = State(0.5 * np.outer(bell10_vect, bell10_vect), [2,2])
bell11 = State(0.5 * np.outer(bell11_vect, bell11_vect), [2,2])
```

Later in Subsection 3.4.5 we shall see other ways to defined these states, namely via state transformation. The same holds for the Greenberger-Horne-Zeilinger (GHZ) states:

```
ghz_vect1 = np.array([1,0,0,0,0,0,0,1])
ghz_vect2 = np.array([1,0,0,0,0,0,0,-1])
ghz_vect3 = np.array([0,0,0,1,1,0,0,0])
ghz_vect4 = np.array([0,0,0,1,-1,0,0,0])
ghz_vect5 = np.array([0,0,1,0,0,1,0,0])
ghz_vect6 = np.array([0,0,1,0,0,-1,0,0])
ghz_vect7 = np.array([0,1,0,0,0,0,1,0])
ghz_vect8 = np.array([0,1,0,0,0,0,-1,0])

ghz1 = State(0.5 * np.outer(ghz_vect1, ghz_vect1), [2,2,2])
ghz2 = State(0.5 * np.outer(ghz_vect2, ghz_vect2), [2,2,2])
ghz3 = State(0.5 * np.outer(ghz_vect3, ghz_vect3), [2,2,2])
ghz4 = State(0.5 * np.outer(ghz_vect4, ghz_vect4), [2,2,2])
ghz5 = State(0.5 * np.outer(ghz_vect5, ghz_vect5), [2,2,2])
ghz6 = State(0.5 * np.outer(ghz_vect6, ghz_vect6), [2,2,2])
ghz7 = State(0.5 * np.outer(ghz_vect7, ghz_vect7), [2,2,2])
ghz8 = State(0.5 * np.outer(ghz_vect8, ghz_vect8), [2,2,2])
```

Finally, the *EfProb* library makes it possible to define qubits (states of dimension 2) via spherical coordinates of the Bloch sphere. For instance the qubits `ket(0)`, `ket(1)`, `plus`, `minus` can alternatively be defined as, respectively:

```
bloch_state(0, 0)
bloch_state(pi, 0)
bloch_state(pi/2, 0)
bloch_state(pi/2, pi)
```

The coordinates (θ, ϕ) of the function `bloch_state`(θ, ϕ) have the ranges: $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi]$. The resulting state is obtained from the vector state $(\cos(\theta/2), \sin(\theta/2) \cdot e^{i \cdot \phi})$.

3.1.1 *Operations on states*

This subsection describes the four operations that can be applied to states themselves, namely *parallel product*, *convex sum*, and *marginalisation*. Later we shall see more operations that produce new states, but they require other structures that we have not yet described, like predicates for conditioning (see Subsection 3.2.3) and channels for state transformation (see Subsection 3.4.2).

Product states As before we can form product states — also called joint states — by putting states in parallel via the binary operator `@`.

```
>>> ket(0) @ ket(1)
[[ 0.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
>>> ket(1) @ plus
[[ 0.0+0.j  0.0+0.j  0.0+0.j  0.0+0.j]
 [ 0.0+0.j  0.0+0.j  0.0+0.j  0.0+0.j]
 [ 0.0+0.j  0.0+0.j  0.5+0.j  0.5+0.j]
 [ 0.0+0.j  0.0+0.j  0.5+0.j  0.5+0.j]]
```

The resulting matrix is the *Kronecker product* of the two matrices. As one sees, the size of these matrices grows quickly. Indeed, the product of an n-dimensional state and an m-dimensional state has dimension $n \times m$.[2]

There is a short-hand `ket(0,0)` for the product `ket(0) @ ket(0)`. An arbitrary tuple of `0`'s and `1`'s can be used as input to the function `ket`:

```
>>> ket(0,1,1)
[[ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]]
```

This is the same as `ket(0) @ ket(1) @ ket(1)`. Using other entries than `0` or `1` yields an exception.

Products can also be iterated as:

```
>>> ket(0) ** 3
[[ 1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]]
```

This is `ket(0) @ ket(0) @ ket(0)`, which is the same as `ket(0,0,0)`. The operators `@` and `**` can be combined, leading quickly to large states like:[3]

```
>>> (plus @ ket(0)) ** 4
>>> ...
```

[2] We have said that a qubit has dimension 2. Each state `s` has a domain attribute called `s.dom`. It returns a list of dimensions of all its component states. For instance for the large example state we can ask for `((plus @ ket(0)) ** 4).dom` and get `[2, 2, 2, 2, 2, 2, 2, 2]`. This domain is used as the *type* of a state, and is checked when states are combined with for instance predicates or channels. The size `s.dom.size` is the product of all numbers in `dom`.

[3] The parallel product operator `@` is 'strictly' associative, so that `(s1 @ s2) @ s3` and `s1 @ (s2 @ s3)` are the same states.

Convex sums of states The second construction on states in this subsection is the convex sum. It involves a weighted sum of states where the weights add up to one, as in:

```
>>> convex_state_sum( (0.2,ket(0)), (0.3,plus), (0.5,minus) )
[[ 0.6+0.j -0.1+0.j]
 [-0.1+0.j  0.4+0.j]]
>>> print( convex_state_sum( (1/2, plus), (1/2, minus) )
[[ 0.5+0.j  0.0+0.j]
 [ 0.0+0.j  0.5+0.j]]
```

The latter is the fair quantum coin state.

Marginalisation Via products one joins states together. Marginalisation is the reverse operation that deletes certain parts of joint product states. How this works is best illustrated via an example:

```
>>> ket(0) @ ket(1) @ plus @ minus % [1,0,1,0]
[[ 0.5+0.j  0.5+0.j  0.0+0.j  0.0+0.j]
 [ 0.5+0.j  0.5+0.j  0.0+0.j  0.0+0.j]
 [ 0.0+0.j  0.0+0.j  0.0+0.j  0.0+0.j]
 [ 0.0+0.j  0.0+0.j  0.0+0.j  0.0+0.j]]
```

The post-fix selection operation `%[1,0,1,0]` selects the first and third component — indicated by `1` in first and third position in the selector `[1,0,1,0]` — from the joint state, and removes the second and fourth component — indicated by `0` in the second and fourth position. What remains in the example is the product state `ket(0) @ plus`. Mathematically this marginalisation operation takes the appropriate partial traces.[4]

Marginalisation exists in a quantum setting because quantum bits/information can be discarded. But since there is no *cloning* in a quantum world, there is no copy operation. As already mentioned in the beginning of this chapter, this is a crucial difference with the world of classical (discrete and continuous) probability.

In general, a joint state `s` is different from the product of its marginals, that is, `s` is not equal to `(s % [1,0]) @ (s % [0,1])`. This is due to possible *entanglement* within `s`. Via such entanglement one part of the state can influence the other part, as we have also seen in classical probability theory, see Example 13. But quantum entanglement is stronger than probabilistic entwinedness.

[4] We have described in sidenote 2 that the domain attribute 'dom' of a state is a list of dimensions of the components states, say `[2,2,4]`, and that such lists are used as types for states. A selector in marginalisation must match such dom attribute, and in this case also be of length three, consisting of `0`'s and `1`'s, describing whether each of these components must be kept or removed. If a marginalisation selector does not match the type of a state, an exception is raised.

3.1.2 *Basic states*

We have already seen basic states like `ket(0)`, `ket(1)`, `plus`, `minus`. This section describes several more functions that produce quantum states, namely:

- probabilistic states
- vector states
- random states

Probabilistic states A quantum state is called *probabilistic* if its all its non-zero matrix entries are on the diagonal. These entries are then necessarily real and add up to one. The states `ket(0)` and `ket(1)` are thus probabilistic, but `plus` and `minus` are not. Probabilistic states are often called *classical* states.

There is function that takes an arbitrary-length tuple of numbers, normalises them, and turns them into a probabilistic state:

```
>>> probabilistic_state(3,4,1)
[[ 0.375  0.     0.   ]
 [ 0.     0.5    0.   ]
 [ 0.     0.     0.125]]
```

There is a special function for the binary case:

```
>>> cflip(0.3)
[[ 0.3  0. ]
 [ 0.   0.7]]
```

where `cflip` stand for 'classical flip'. It yields the same outcome as `probabilistic_state(0.3, 0.7)` and as `probabilistic_state(3, 7)`. There are further abbreviations `cfflip` for `cflip(0.5)`, giving a classical 'fair' flip. These are just different names for `ket(0)` and `ket(1)`.

The product and convex sum of probabilistic states is again probabilistic. For instance:

```
>>> cfflip @ probabilistic_state(2,2,1)
[[ 0.2  0.   0.   0.   0.   0. ]
 [ 0.   0.2  0.   0.   0.   0. ]
 [ 0.   0.   0.1  0.   0.   0. ]
 [ 0.   0.   0.   0.2  0.   0. ]
 [ 0.   0.   0.   0.   0.2  0. ]
 [ 0.   0.   0.   0.   0.   0.1]]
```

Vector states Each vector $|v\rangle$ of norm 1 yields a state / density matrix $|v\rangle\langle v|$, which is often called a vector state. Thus, by provinding an n-tuple of complex numbers we can get a state. Implicitly, the numbers are normalised:

```
>>> vector_state(1, complex(1, 2), -2)
[[ 0.1+0.j   0.1-0.2j -0.2-0.j ]
 [ 0.1+0.2j  0.5+0.j  -0.2-0.4j]
 [-0.2+0.j  -0.2+0.4j  0.4+0.j ]]
```

In the 2-dimensional case such a vector $|v\rangle$ is of the form $|v\rangle = \binom{\alpha}{\beta}$, where $\alpha, \beta \in \mathbb{C}$ are complex numbers with $|\alpha|^2 + |\beta|^2 = 1$. The resulting density matrix is:

$$\begin{pmatrix} |\alpha|^2 & \alpha \cdot \overline{\beta} \\ \overline{\alpha} \cdot \beta & |\beta|^2 \end{pmatrix}$$

```
>>> vector_state(0.5 * sqrt(3), complex(0, 0.5))
[[ 0.75+0.j         0.00-0.4330127j]
 [ 0.00+0.4330127j  0.25+0.j       ]]
```

There is a general result saying that each (finite-dimensional) density matrix can be written as a convex combination (a 'mixture') of 'pure' vector states, via spectral decomposition, see *e.g.* Thm. 2.5 of[5].

[5] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000

A *unit state* is a special case of a vector state — and also of a probabilistic state. It puts a single 1 at the indicated position of the diagonal:

```
>>> point_state(2,3)
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  1.]]
```

This is the same as `vector_state(0,0,1)`.

Random states For testing purposes it is very useful to have randomly generated states of a certain length. For instance:

```
>>> random_state([3])
[[ 0.32162863+0.j          0.23537749-0.07970437j -0.08834685-0.11437522j]
 [ 0.23537749+0.07970437j  0.49432253+0.j         -0.08974455+0.06699254j]
 [-0.08834685+0.11437522j -0.08974455-0.06699254j  0.18404884+0.j        ]]
```

Each time this function is called a new random state is produced, using the underlying random number generator of python.

There is a similar function that produces a random probabilistic state, as in:

```
>>> random_probabilistic_state([2])
[[ 0.59979369  0.        ]
 [ 0.          0.40020631]]
```

Earlier, in the beginning of Section 3.1, we already described the Bell and GHZ states, essentially as vector states.

3.2 *Predicates*

The distinction between states and predicates is fundamental in our approach to probability theory, both in the classical and in the

quantum case. Having seen quantum states, we now introduce quantum predicates. In combination with states we can obtain the *validity* of a predicate in a state, and also the state *conditioned* by a predicate. But first we shall describe predicates on their own, with several ways to combine them into new predicates.

Mathematically, a predicate p of dimension n is an $n \times n$ matrix $p \in \mathsf{M}_n$ over the complex numbers, which is positive and below the identity: $0 \leq p \leq 1$, there $0, 1 \in \mathsf{M}_n$ are the zero matrix and the identity matrix respectively. These quantum predicates are often called *effects*. A predicate p is called *sharp* if $p \cdot p = p$. The matrices of predicates may involve complex numbers, as for instance in:

$$\frac{1}{2}\begin{pmatrix} 1 & -i \\ i & 1 \end{pmatrix}$$

If $\omega \in \mathsf{M}_n$ is a state and $p \in \mathsf{M}_n$ is a predicate, then the validity $\omega \models p$ of p in ω is defined as the number in the unit interval $[0, 1]$ defined as:

$$\omega \models p \;=\; \mathsf{tr}(\omega \cdot p), \tag{3.1}$$

where tr is the trace operation and $\cdot$ is matrix composition. This equation is also known as Born's rule.

The most basic predicates are truth and falsity:

```
>>> falsity([2])
[[ 0.  0.]
 [ 0.  0.]]
>>> truth([2,3])
[[ 1.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  1.]]
```

Notice that the arguments of `falsity` and `truth` are not single dimensions but lists of dimensions. These domains describe the type of a state, see sidenote 2, and make it possible to define truth and falsity on an entire product state at once, by using the appropriate domain of the product state as input. Thus, the predicate `truth(2,3)` used above is the same as the parallel product `truth([2]) @ truth([3])` that will be described below.

A *probabilistic* predicate has, like a probabilistic state, only real entries from $[0, 1]$ on its diagonal, but these entries need not add up to one. There is a function that produces such a probabilistic predicate of dimension n from an n-tuple of numbers in the unit interval.

```
>>> probabilistic_pred(0.1,1,0.25,0.0)
[[ 0.1   0.    0.    0.  ]
 [ 0.    1.    0.    0.  ]
 [ 0.    0.    0.25  0.  ]
 [ 0.    0.    0.    0.  ]]
```

As a special case there are 'point' predicates that put a `1` at the given position of the diagonal:

```
>>> point_pred(2,4)
[[0 0 0 0]
 [0 0 0 0]
 [0 0 1 0]
 [0 0 0 0]]
```

The analogues of the classical 'yes' and 'no' predicates have dimension two:

```
>>> yes_pred = point_pred(0,2)
>>> no_pred = point_pred(1,2)
```

For testing purposes one can use `random_pred([n1,..nk])` and `random_probabilistic_pred([n1,..,nk])` to get arbitrary (probabilist) predicates of dimension `n1` $\times\cdots\times$ `nk`.

Each state of dimension n is also a predicate of dimension n, but not the around — take `falsity` for instance. There is a way in *EfProb* to turn a state into a predicate, via the method `as_pred()`, but it will not be used frequently. Since the distinction between predicates and states is so fundamental, we prefer to define predicates immediately as predicates, and not first as states. For this reason there is a function `vector_pred`, in analogy with `state_pred`. It turns a vector $|v\rangle$ into the outer product $|v\rangle\langle v|$, considered as a predicate.

Again, it is important not to confuse states and predicates. They are very different. As we shall see next, there are several logical operations on predicates which do not exists for states, in the sense that states are not closed under these operations.

3.2.1 *Operations on predicates*

We shall discuss the following operations on predicates.

- orthosupplement, or negation `~`
- scalar multiplication `*`
- partial sum `+`
- parallel conjunction `@`
- sequential conjunction `&`

- weakening.

The first three points capture the *effect module* structure of quantum predicates. This effect module structure is common for predicates in discrete, continuous, and quantum probability. They are used for instance in logical formulas in (quantum) cognition (see[6]).

[6] B. Jacobs. Quantum effect logic in cognition. *Journ. Math. Psychology*, 81: 1–10, 2017b. See https://doi.org/10.1016/j.jmp.2017.08.004

Orthosupplement Negation in a probabilistic or quantum setting is called orthosupplement. Here it is written as $\sim p$, but in the literature one often sees $p^{\perp}$ instead. On the unit interval $[0,1]$ orthosupplement is given by $\sim r = 1 - r$. In a quantum setting the orthosupplement of a predicate is given similarly, by subtracting the predicate from the identity (diagonal) matrix. Thus, $\sim p = 1 - p$, see:

```
>>> p = random_pred([2])
>>> p
[[ 0.36250803+0.j         -0.15404606-0.00875401j]
 [-0.15404606+0.00875401j  0.96265550+0.j        ]]
>>> ~p
[[ 0.63749197+0.j          0.15404606+0.00875401j]
 [ 0.15404606-0.00875401j  0.03734450+0.j        ]]
```

Notice that the orthosupplement of a probabilistic predicate is probabilistic again. Obviously, taking the orthosupplement twice returns the original predicate. Hence the 'double negation' rule of classical (Boolean) logic holds. But the logic of quantum/probabilistic predicate is not Boolean. Its structure is captured algebraically in terms of effect modules.

Scalar multiplication Predicates are closed under multiplication with a scalar from the unit interval $[0,1]$. Each element of the matrix is then multiplied by the scalar. Here is a simple probabilistic example.

```
>>> 0.3 * probabilistic_pred(0.2,0.5)
[[ 0.06  0.  ]
 [ 0.    0.15]]
```

Probabilistic predicates are closed under scalar multiplication, but sharp predicates are not.

Partial sum Addition of numbers $r, s \in [0,1]$ in the unit interval is a partial operation, which is defined only if $r + s \leq 1$. This sum is commutative and associative in a suitable partial sense. Similarly there is a partial sum of predicates, which is given by (elementwise) sum of matrices, as long as the resulting matrix is still below the identity (matrix). Here is an example.

```
>>> point_pred(2,4) + point_pred(0,4)
[[1 0 0 0]
 [0 0 0 0]
 [0 0 1 0]
 [0 0 0 0]]
```

There is a predefined x-predicate. We show what its orthosupplement is:

```
>>> x_pred
[[ 0.5 -0.5]
 [-0.5  0.5]]
>>> p = vector_pred(1/sqrt(2), 1/sqrt(2))
>>> p
[[ 0.5  0.5]
 [ 0.5  0.5]]
>>> x_pred + p
[[ 1.  0.]
 [ 0.  1.]]
```

It is not hard to see that a convex sum of predicates always exists. Such sums can be written simply as `0.3 * p1 + 0.6 * p2 + 0.1 * p3` for predicates `p1, p2, p3`. This works in the same way for n-ary convex sums, for any fixed $n \geq 1$.

A *test* is a finite list of predicates `p1`, ..., `pn`, all with the same domain, whose sum `p1 + ... + pn` equals truth. Such tests are used in measurements, see Subsection 3.4.6. The bell and GHZ basis vectors mentiond at the end of Section 3.1 give rise to the following Bell and GHZ tests, in which these states are used as predicates.

```
>>> bell_test = [bell00.as_pred(), bell01.as_pred(),
...              bell10.as_pred(), bell11.as_pred()]
>>> ghz_test = [ghz1.as_pred(), ghz2.as_pred(),
...             ghz3.as_pred(), ghz4.as_pred(),
...             ghz5.as_pred(), ghz6.as_pred(),
...             ghz7.as_pred(), ghz8.as_pred()]
```

Indeed, the Bell predicates form a test (and the GHZ predicates too):

```
>>> bell00.as_pred() + bell01.as_pred() + bell10.as_pred() + bell11.as_pred()
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

When the matrix sum $p + q$ of two predicates p, q is not below the identity, their sum still exists in *EfProb*, but it no longer a predicate. It has become a random variable, see Section 3.3 for details.

Parallel conjunction If `p1` is a predicate on state `s1` and `p2` is a predicate on `s2`, then there is a parallel conjunction predicate `p1 @ p2` on the product state `s1 @ s2`. It is obtained via the Kronecker product of the two matrices, just like for states. The predicate `p1 @ p2` is like a multiplication, see *e.g.* in:

```
>>> truth([2]) @ falsity([3])
[[ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]]
```

In this way the underlying matrices grow considerable in size.[7]

Sequential conjunction For two predicates `p`, `q` of the same type, there is a predicate `p & q` for sequential conjunction. It is pronounced as '`p` and-then `q`'. The order is relevant, since this conjunction `&` is *not* commutative in the quantum case — unlike in classical probability.

The mathematical interpretation of 'and-then' is given by:

$$p \,\&\, q \;=\; \sqrt{p} \cdot q \cdot \sqrt{p}. \tag{3.2}$$

Here we use $\sqrt{p}$ as the square root of the *matrix* `p`.[8] It exists since p, as a predicate, is positive. This construction is sometimes called *Lüders rule*. It's role will be illustrated in Example 24. Here we just use a probabilistic example, where sequential conjunction `&` is simply pointwise multiplication — and thus commutative.

```
>>> probabilistic_pred(0.2,0.8) & probabilistic_pred(0.4, 0.6)
[[ 0.08  0.  ]
 [ 0.    0.48]]
```

We shall see illustrations of non-commutativity of & later on, for instance in Example 24.

Weakening Weakening is a structural rule in logic that makes it possible to "widen the context", see the general explanation at the end of Subsection 1.2.2 for discrete probability. It works the same way in quantum probability: we have to deal with weakening explicitly, namely via parallel conjunction with truth:

```
>>> t = ket(0,0)
>>> p = point_pred(1,2)
>>> s >= p
0.6
>>> s @ t >= p @ truth(t.dom)
0.6
```

[7] When we take the sum `p+q` of two predicates, these predicates `p` and `q` have to have *the same domain*. This domain of a predicate is the same as the domain of a state, as sketched in sidenote 2: it is given by a list of natural numbers describing the dimensions of their components. These dimension lists (domains) are simply concatenated in the operation `@`, both for states and for predicates. We often informally say things like "`p` is a predicate on states `s`". This means that `p` and `s` have the same domain/type, as given by their `dom` attribute.

[8] The square root of a matrix is not the same thing as the square root of its elements separately. The standard way to obtain the square root of a positive (*i.e.* non-negative, sometimes called positive definite) matrix M is to diagonalise M as $M = VEV^*$, where E is a diagonal matrix containing the (positive) eigenvalues of M. Let D be the diagonal matrix with the square roots of the eigenvalues in E, taken elementwise. Then $\sqrt{M} = VDV^*$.

3.2.2 *Validity*

As we already mentioned in the beginning of Section 3.2, validity $\omega \models p$ of a predicate p in a state ω is a very basic notion. In Python the operator that is most similar to '$\models$' is `>=`. Hence this what we shall use. Equation (3.1) describes $\omega \models p$ as the trace $\mathrm{tr}(\omega \cdot p)$ of the product. It can be understood as the Hilbert-Schmidt inner product $\langle \omega \,|\, p \rangle$, using that $p^* = p$.

```
>>> v = vector_pred(0.5 * sqrt(3), complex(0, 0.5))
>>> v
[[ 0.75+0.j         0.00-0.4330127j]
 [ 0.00+0.4330127j  0.25+0.j       ]]
>>> ket(0) >= v
0.75
>>> ket(1) >= v
0.25
```

Validity $\models$ satisfies some basic mathematical properties:

$$\begin{aligned}
\omega \models \mathsf{truth} &= 1 \\
\omega \models \mathsf{falsity} &= 0 \\
\omega \models \sim p &= 1 - (\omega \models p) \\
\omega \models r * p &= r * (\omega \models p) \\
\omega \models p_1 + p_2 &= (\omega \models p_1) + (\omega \models p_2) \\
\omega \mathbin{@} \sigma \models p \mathbin{@} q &= (\omega \models p) \cdot (\sigma \models q).
\end{aligned}$$

Such properties can be tested (but not proven!) by using random states and predicates, as in:

```
>>> s = random_state([100])
>>> p = random_pred([100])
>>> s >= 0.3 * p
0.05918743186783347
>>> 0.3 * (s >= p)
0.059187431867833463
```

Using validity we can now easily see that sequential conjunction & is *not commutative*:

```
>>> p = point_pred(0,2)
>>> q = plus.as_pred()
>>> ket(0) >= p & q
0.5
>>> ket(0) >= q & p
0.25
```

See Example 24 for more such examples.

Since a state can be turned into a predicate, we can write states on both sides of the validity operator `>=`, where we do need to turn the

state on the right hand side into a predicate. The resulting validity does not depend on which of the two states is used as predicate.

```
>>> s = random_state([8])
>>> t = random_state([8])
>>> s >= t.as_pred()
0.116791838672
>>> t >= s.as_pred()
0.116791838672
```

If these states `s` and `t` are vector states, then validity corresponds to the square of the inner product of the underlying states, as tested in:

```
>>> v1 = np.random.rand(5)
>>> v2 = v1/np.linalg.norm(v1)
>>> w1 = np.random.rand(5)
>>> w2 = w1/np.linalg.norm(w1)
>>> s = vector_state(*v2)
>>> t = vector_state(*w2)
>>> s >= t.as_pred()
0.659104940735
>>> np.inner(v2, w2) ** 2
0.659104940735
```

Example 23 *The Bell states are famous 2-qubit states which have been defined at the end of Section 3.1. Of these four states we use the one named* `bell00`*. We additionally use the following predicates, obtained from states.*

```
>>> A1 = vector_pred(1/sqrt(2), 1/sqrt(2))
>>> B1 = A1
>>> A2 = vector_pred(1/sqrt(2), 0.5/sqrt(2) * complex(1, -sqrt(3)))
>>> B2 = A2
>>> A1
[[ 0.5  0.5]
 [ 0.5  0.5]]
>>> A2
[[ 0.50+0.j         0.25+0.4330127j]
 [ 0.25-0.4330127j  0.50+0.j       ]]
```

These predicates may look a bit ad hoc *at this stage,*[9] *but later in Subsection 3.4.3 we show how they can be described in a more systematic manner via predicate transformation.*

The valdities of the parallel conjunctions of the predicates `A1`, `A2`, `B1`, `B2` *and their orthosupplements yield the famous Bell-tables that play an important role in non-locality in quantum foundations, see e.g.* [10] *for more information. The table below summarises the validities of the 4-tuples of predicates, forming a test:*

```
Ai @ Bj, Ai @ ~Bj, ~Ai @ Bj, ~Ai @ ~Bj
```

[9] Detailed calculations and explanations of the coice of these predicates can be found in [Abramsky, 2014].

[10] S. Abramsky and A. Brandenburger. The sheaf-theoretic structure of non-locality and contextuality. *New Journ. of Physics*, 13:113036, 2011

for $i, j \in \{1, 2\}$.

`bell00 >= A1 @ B1`	`bell00 >= A1 @ ~B1`	`bell00 >= ~A1 @ B1`	`bell00 >= ~A1 @ ~B1`
0.5	*-1.11022302463e-16*	*-1.11022302463e-16*	*0.5*
`bell00 >= A1 @ B2`	`bell00 >= A1 @ ~B2`	`bell00 >= ~A1 @ B2`	`bell00 >= ~A1 @ ~B2`
0.375	*0.125*	*0.125*	*0.375*
`bell00 >= A2 @ B1`	`bell00 >= A2 @ ~B1`	`bell00 >= ~A2 @ B1`	`bell00 >= ~A2 @ ~B1`
0.375	*0.125*	*0.125*	*0.375*
`bell00 >= A2 @ B2`	`bell00 >= A2 @ ~B2`	`bell00 >= ~A2 @ B2`	`bell00 >= ~A2 @ ~B2`
0.125	*0.375*	*0.375*	*0.125*

3.2.3 *Conditioning*

Validity $\omega \models p$ is one way to combine states and predicates. When this validity is non-zero, one can also form the conditional state $\omega|_p$, which is written in Python as ω/p. This new state is an update (revision) of ω given the information given by the predicate p.

Mathematically this conditioning is given by the formula:

$$\omega|_p = \frac{\sqrt{p} \cdot \omega \cdot \sqrt{p}}{\omega \models p}. \tag{3.3}$$

The use of square root matrices reminds us of sequential conjunction &, see (3.2). Indeed, there is a close connection given by Bayes' rule:

$$\omega|_p \models q = \frac{\omega \models p \,\&\, q}{\omega \models q}. \tag{3.4}$$

Here is a test of Bayes rule in the quantum world:

```
>>> s = random_state([2])
>>> p = random_pred([2])
>>> q = random_pred([2])
>>> s / p >= q
0.5479352799030488
>>> (s >= p & q) / (s >= p)
0.5479352799030488
```

Conditional states are a powerful primitive, as will be illustrated next.

Example 24 *Polarised light filters are a great illustration of the 'weirdness' of the quantum world. Briefly, if we first polarise a light source vertically as* ↕*, and then check how much horizontally polarised light* ↔ *is visible, we see nothing, of course. But if one puts a diagonally oriented filter* ⤢ *inbetween* ↕ *and* ↔*, light does go trough the sequence of filters* ↕, ⤢, ↔*, namely one quarter of it.*

Here we describe this situation via condtioning of states, following [11]*. For more background information see e.g.* [12]*.*

We start with a random qubit state, and the relevant predicates.

```
>>> s = random_state([2])
>>> vert_filt = point_pred(0,2)
>>> hor_filt = point_pred(1,2)
>>> diag_filt = plus.as_pred()
>>> s >= vert_filt
0.574918520238
>>> s_vert = s / vert_filt
```

The latter probability tells how much of the light from the arbitrary source (state) is horizontally polarised. We proceed to condition the state. This corresponds to putting a horizontally polarised filter right after the (random) source:

```
>>> s_vert >= vert_filt
1.0
>>> s_vert >= hor_filt
0.0
```

Not surprisingly, after this vertical polarisation all of the light is vertically polarised, but none of it is horizontally polarised.

Next we ask for the probability of light coming through if we first look at the vertically polarised light diagonally and then *horizontally:*

```
>>> s_vert >= diag_filt & hor_filt
0.25
```

Now (approximately) one quarter is horizontally polarised! If we do this in the other order, we still see nothing:

```
>>> s_vert >= hor_filt & diag_filt
0.0
```

This shows the non-commutativity of sequential conjunction &*.*

If we enforce diagonal polarisation via a second filter — that is, via conditioning once more — and then ask how much of the light is horizontally polarised we get a probability of one half:

```
>>> s_vert / diag_filt >= hor_filt
0.5
```

[11] B. Jacobs. New directions in categorical logic, for classical, probabilistic and quantum logic. *Logical Methods in Comp. Sci.*, 11(3):1–76, 2015

[12] E. Rieffel and W. Polak. *Quantum Computing. A Gentle Introduction*. MIT Press, Cambridge, MA, 2011

To conclude, we use this polarisation example for some counter examples. First, we show that iterated conditioning yields a different outcome than conditioning with conjunction:

```
>>> s / vert_filt / diag_filt
[[ 0.5+0.j  0.5+0.j]
 [ 0.5+0.j  0.5+0.j]]
>>> s / (vert_filt & diag_filt)
[[ 1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j]]
```

In classical probability these outcomes are the same.

This polarisation example can also be used to illustrate the failure of the law of total probability in the quantum setting, see also Example 7. For instance:

```
>>> s >= hor_filt
0.425081479762
```

The 'law of total probability' formulation gives a different outcome:

```
>>> (s / diag_filt >= hor_filt) * (s >= diag_filt) +
...    (s / ~diag_filt >= hor_filt) * (s >= ~diag_filt)
0.5
```

Since Baye's rule (3.4) does hold in the quantum world, the latter probability can also be obtained as:

```
>>> (s >= diag_filt & hor_filt) + (s >= ~diag_filt & hor_filt)
0.5
```

Finally, we show that sequential conjunction & *is not only non-commutative, but also* non-associative*:*

```
>>> s / vert_filt >= vert_filt & (diag_filt & hor_filt)
0.25
>>> s / vert_filt >= (vert_filt & diag_filt) & hor_filt
0.0
```

The following example contains the essence of what has become known as the EPR paradox[13]. It involves the influence of conditioning in one component on the other component of an entangled joint state. The analogous phenomenon in classical probability has already been described in Example 6 — with the same predicates, resulting in the same probabilities.

[13] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Physical Review*, 47(10): 777–780, 1935

Example 25 *We consider the Bell state* `bell00`*, and are interested in the validity of the 'zero' point predicate in the second component. We have to use weakening via the truth predicate in the first components to make the types match.*

```
>>> bell00 >= truth([2]) @ point_pred(0, 2)
0.5
```

Conditioning in the first component changes the validity of this predicate in the second component.

```
>>> bell00 / (point_pred(0, 2) @ truth([2])) >= truth([2]) @ point_pred(0, 2)
1.0
>>> bell00 / (point_pred(1, 2) @ truth([2])) >= truth([2]) @ point_pred(0, 2)
0.0
```

This is what Einstein called spukenhaften Fernwirkung *(spooky interaction at a distance).*

This same crossover influence is often illustrated with 'X' predicates `x_pp` *and* `x_mp`*, where '*`pp`*' and '*`mp`*' stand for 'plus predicate' and 'min predicate'. They arise via the eigenvectors of the X matrix* $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$.

```
>>> x_pp = vector_pred(-1/sqrt(2), 1/sqrt(2))
>>> x_mp = vector_pred(1/sqrt(2), 1/sqrt(2))
>>> bell00 >= truth([2]) @ x_plus
0.5
>>> bell00 / (x_pp @ truth([2])) >= truth([2]) @ x_pp
1.0
>>> bell00 / (x_mp @ truth([2])) >= truth([2]) @ x_pp
0.0
```

The same validities appear if one uses the state `bell01` *instead of* `bell00`*. For the states* `bell10` *and* `bell11` *the last two probabilities are exchanged.*

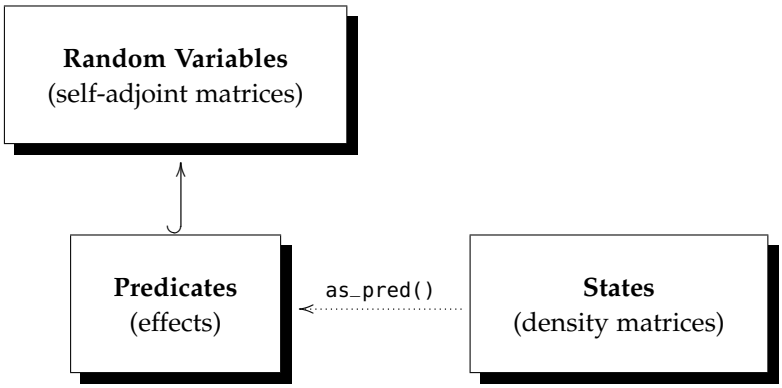
This 'crossover' influence can be described more explicitly via conditioning in one coordinate and then projecting (marginalising) that coordinate away:

```
>>> (bell00 / (point_pred(0,2) @ truth([2]))) % [0,1] == point_state(0,2)
True
>>> (bell00 / (point_pred(1,2) @ truth([2]))) % [0,1] == point_state(1,2)
True
```

3.3 Random variables

So far we have seen states and predicates. States are density matrices: positive square matrices (operators) whose trace is one. Predicates are effects: positive square matrices below the identity. States can be turned into predicates via their `as_pred()` method. In the *EfProb* library predicates form a subclass, namely of random variables. Mathematically the latter are *self-adjoint* square matrices: they are equal to their conjugate transpose. Such operators are also called *Hermitian*. Thus we have the following structure of classes in Python,

corresponding to inclusions between sets of associated matrices.



These conversions from predicates to random variables is performed implicitly. Predicates are closed under scalar multiplication with a number from the unit interval. Random variables are closed under scalar multiplication with arbitrary real — but not complex — numbers. Hence we get the following type information:

```
>>> type( ket(0) )
<class 'efprob_qu.State'>
>>> type( 0.5 * ket(0).as_pred() )
<class 'efprob_qu.Predicate'>
>>> type( 5 * ket(0).as_pred() )
<class 'efprob_qu.RandVar'>
```

In discrete and continuous probability a predicate on a sample space Ω is a function $\Omega \to [0,1]$. A random variable is a more general function $\Omega \to \mathbb{R}$ to the real numbers. This difference is reflected in the quantum case, in the following way. The eigenvalues of an effect lye in the unit interval $[0,1]$. And eigenvalues of a self-adjoint matrix are real-valued.

3.3.1 Operations on random variables

Random variables are closed under addition `+`, subtraction `-`, negation `-` and scalar multiplication $*$ with a real number. These operations are applied pointwise to all the elements of the matrices involved. Hence these basic arithmetical operations can be applied more generally than to predicates, where, for instance, addition `+` is only partially defined. If the sum of two predicates does not exist, as predicate, the result is automatically cast to a random variable:

```
>>> type( point_pred(0,2) )
<class 'efprob_qu.Predicate'>
>>> type( point_pred(0,2) + point_pred(0,2) )
<class 'efprob_qu.RandVar'>
```

Random variables have a domain like predicates and states. Random variable can be put in parallel via `@`. In fact, in the *EfProb* library, parallel product `@` is defined for the superclass of random variable, and then inherited by the subclass of predicates.

Validity `s >= rv` in state `s` is defined for a random variable `rv` and not just for a predicate. This validity is what is usually called the expectation. But, in contrast, conditioning `s / p` of a state `s` is defined only for a predicate `p`, and not for a random variable. These types are checked in the *EfProb* implementation. Also, the sequential conjunction `&` and disjunction `|` operations on predicates do not exist for random variables. They exist only for predicates.

A new operation that exists on random variables is *variance*. We shall illustrate it for an ordinary dice, incorporated into the quantum world.

```
>>> dice = uniform_probabilistic_state([6])
>>> points = 1 * point_pred(0,6) + 2 * point_pred(1,6) + 3 * point_pred(2,6) \
...          + 4 * point_pred(3,6) + 5 * point_pred(4,6) + 6 * point_pred(5,6)
>>> dice >= points
3.5
>>> dice.variance(points)
2.91666666667
```

Implicitly in the defintion of `point` the predicates `unit_pred` are cast to random variables, via scalar multiplication and addition.

(The expectation and variance values computed above in the quantum framework coincide with the classical ones, see the end of Subsection 1.3.1.)

We conclude this section with the famous Bell inequalities. As we shall see, they involve (validities of) random variables, and not predicates.

Example 26 *We show how to obtain the Bell (actually 'CHSC') inequality in EfProb, where we closely follow the description in*[14]*. We start with x/y/z random variables, obtained from the corresponding predicates.*

[14] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000

```
>>> x_rv = ~x_pred - x_pred
>>> y_rv = ~y_pred - y_pred
>>> z_rv = z_pred - ~z_pred
```

They are used to define the following four random variables:

```
>>> Q = z_rv
>>> R = x_rv
>>> S = 1/sqrt(2) * (-z_rv - x_rv)
>>> T = 1/sqrt(2) * (z_rv - x_rv)
```

These random variables `Q`, `R`, `S`, `T` *are really random variables and not predicates: they all have eigenvalues* -1 *and* 1*, lying outside the unit interval* $[0,1]$*, as must be the case for predicates.*

One then considers validities of suitable parallel conjunctions in one of the Bell states.

```
>>> psi = bell11
>>> psi >= Q @ S; psi >= R @ S; psi >= R @ T; psi >= Q @ T
0.707106781187
0.707106781187
0.707106781187
-0.707106781187
>>> 2 < (psi >= Q @ S + R @ S + R @ T - Q @ T)
True
```

The latter forms the famous violation. Notice that it used parallel conjunction `@`*, since in the corresponding physical model, measurements are performed in parallel, see*[15] *for details.*

As an aside, these random variables can be used to show that parallel composition is not monotone in its first coordinate. The random variable `Q` *is below the identity (truth), since:*

```
>>> is_positive(truth([2]).array - Q.array)
True
```

But:

```
>>> (psi >= Q @ S) > (psi >= truth([2]) @ S)
True
```

Other random variables can be used to obtain Bell/CHSH violations. For instance:

```
>>> A1 = -x_rv
>>> A2 = -y_rv
>>> B1 = 1/sqrt(2) * (x_rv + y_rv)
>>> B2 = 1/sqrt(2) * (x_rv - y_rv)
>>> psi >= A1 @ B1 + A1 @ B2 + A2 @ B1 - A2 @ B2
2.82842712475
```

3.4 Channels

States and predicates are nice, but the real action comes from *channels*, also known as quantum operations. They can be used in two directions, namely as mapping

- states to states, as *state transformers*, according to 'Schrödinger's picture';
- predicates to predicates, as *predicate transformers*, according to 'Heisenberg's picture'.

Here we follow the latter approach, so that channels are morphisms in the opposite of the category of von Neumann algebras, see[16] for

[15] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000

[16] K. Cho, B. Jacobs, A. Westerbaan, and B. Westerbaan. An introduction to effectus theory. see arxiv.org/abs/1512.05813, 2015

details.

Mathematically, in the finite-dimensional case, a channel is a completely positive normal linear map between algebras of square matrices. In our implementation the size of these matrices is determined by the 'types' of the domain and codomain of the channel. These types are lists of natural numbers, like for states and for predicates, describing the dimensions of the product components involved. The types are checked at run-time. We can write a channel as an arrow:

$$[n_1, \ldots, n_a] \xrightarrow{\ c\ } [m_1, \ldots, m_b]$$

The type $[n_1, \ldots, n_a]$ of the domain of the channel can be accessed via the `dom` attribute of a channel. Similarly, the type $[m_1, \ldots, m_b]$ is available as `cod` attribute. The channel is represented in *EfProb* as a *nested* matrix, given by:

- an $m \times m$ matrix, where $m = m_1 \cdot \ldots \cdot m_b$, whose entries are:
- $n \times n$ matrices, where $n = n_1 \cdot \ldots \cdot n_a$.

Pictorially this $m \times m$ matrix of $n \times n$ matrices can be seen as:

(3.5)

$$\left(\begin{array}{ccc} \left(\boxed{n \times n}\right) & \cdots & \left(\boxed{n \times n}\right) \\ \vdots & & \vdots \\ \left(\boxed{n \times n}\right) & \cdots & \left(\boxed{n \times n}\right) \end{array}\right) \begin{array}{l} \uparrow \\ m \\ \downarrow \end{array}$$

$$\leftarrow m \rightarrow$$

This matrix can be accessed via an attribute, as in `c.array` for a channel `c`.

The channel implementation involves quite a bit of bookkeeping to handle these matrices appropriately, but luckily an *EfProb* user doesn't have to bother about these details. What has to be kept in mind however, are the types of these channels. For instance, there is *sequential* composition $*$ of channels, which only works if there is a codomain-domain match in the middle, as in:

$$[n_1, \ldots, n_a] \xrightarrow{\ c\ } [m_1, \ldots, m_b] \xrightarrow{\ d\ } [k_1, \ldots, k_c]$$

There is also *parallel* composition `@` of channels. It combines two channels:

$$[n_1, \ldots, n_a] \xrightarrow{\ c\ } [m_1, \ldots, m_b] \qquad [n'_1, \ldots, n'_c] \xrightarrow{\ c'\ } [m'_1, \ldots, m'_d]$$

into a new channel:

(3.6)
$$[n_1, \ldots, n_a, n'_1, \ldots, n'_c] \xrightarrow{\ c \mathbin{@} c'\ } [m_1, \ldots, m_b, m'_1, \ldots, m'_d]$$

Semantically this works via a Kronecker product of the matrices of the two channels.

In addition for a state s and a predicate p we shall define state transformation $c \gg s$ and predicate transformation $c \ll p$, where the types must match, as suggested in the following diagram.

$$\begin{array}{ccc} s & \longmapsto & c \gg s \\ [n_1, \ldots, n_a] & \xrightarrow{\;c\;} & [m_1, \ldots, m_b] \\ c \ll p & \longmapsto & p \end{array} \tag{3.7}$$

Thus:

- state transformation works in the forward (channel) direction: it turns a state s with the same type as the domain of a channel c into a state $c \gg s$ with the same type as the codomain;
- predicate transformation works in the opposite direction: it turns a predicate p with the same type as the codomain into a predicate $c \ll p$ with the same type as the domain.

State- and predicate-transformation interact with validity in the following pleasing manner:

$$c \gg s \models p \;=\; s \models c \ll p. \tag{3.8}$$

This is sometimes called the transformation validity equation.[17]

[17] Essentially, $c \gg (-)$ and $c \ll (-)$ are (bounded) linear operators between Hilbert spaces of operators, with the Hilbert-Schmidt inner product $\langle A \,|\, B \rangle = \mathsf{tr}(A^\dagger B)$. The equation (3.8) says that $c \gg (-)$ and $c \ll (-)$ are each other's daggers $(-)^\dagger$.

3.4.1 *Channels from isometric matrices*

An $m \times n$ complex matrix U is called an *isometry* if its conjugate transpose $U^\dagger$ is its left inverse, that is, if $U^\dagger U = \mathrm{id}$. Such matrices are also called 'dagger monos'. To prevent confusion, $m \times n$ matrices have m rows and n columns, and gives rise to a linear function $\mathbb{C}^n \to \mathbb{C}^m$. We show how it yields a quantum channel $[n] \to [m]$, given by a 'Choi' matrix of the following form:

$$\begin{pmatrix} U^\dagger E_{00} U & \cdots & U^\dagger E_{(m-1)0} U \\ \vdots & & \vdots \\ U^\dagger E_{0(m-1)} U & \cdots & U^\dagger E_{(m-1)(m-1)} U \end{pmatrix} \qquad \text{where} \qquad E_{ij} = |i\rangle\langle j|, \text{ for } i, j < m.$$

The matrices $E_{ij} = |i\rangle\langle j|$ have a 1 at position (i, j) and zero's everywhere else. They form the basis vectors of the vector space M_m of $m \times m$ matrices. The above $m \times m$ matrix thus consists of $n \times n$ matrices $U^\dagger E_{ij} U \colon \mathbb{C}^n \to \mathbb{C}^m \to \mathbb{C}^m \to \mathbb{C}^n$.[18]

In this way the *EfProb* library defines a number of standard channels, mostly from unitary matrices. The latter are matrices U whose

[18] One can form channels more generally, using the Kraus form, via matrices U_i with $\sum_i U_i^\dagger U_i = \mathrm{id}$, but this generality is not used.

conjugate transpose $U^\dagger$ is its inverse, so that both equations $U^\dagger U = \mathrm{id}$ and $UU^\dagger = \mathrm{id}$ hold. This means that both U and $U^\dagger$ are isometries, and can thus be used form channels, in opposite directions, as described above. Such channels from unitaries are sometimes called 'pure'. We reproduce some examples from the *EfProb* library:

```
x_chan = channel_from_isometry(np.array([[0,1],
                                         [1,0]]), [2], [2])
y_chan = channel_from_isometry(np.array([[0,-complex(0, 1)],
                                         [complex(0,1),0]]), [2], [2])
z_chan = channel_from_isometry(np.array([[1,0],
                                         [0,-1]]), [2], [2])
hadamard = channel_from_isometry((1/sqrt(2)) * np.array([ [1, 1],
                                                          [1, -1] ]), [2], [2])
def phase_shift(angle):
    return channel_from_isometry(np.array([[1, 0],
                                           [0, complex(cos(angle),
                                                       sin(angle))]]), [2], [2])
```

These are all channels $[2] \to [2]$.

In addition, we have the following channels $[2,2] \to [2,2]$.

```
cnot = channel_from_isometry(np.array([ [1, 0, 0, 0],
                                        [0, 1, 0, 0],
                                        [0, 0, 0, 1],
                                        [0, 0, 1, 0] ]), [2,2], [2,2])
swap = channel_from_isometry(np.array([ [1, 0, 0, 0],
                                        [0, 0, 1, 0],
                                        [0, 1, 0, 0],
                                        [0, 0, 0, 1] ]), [2,2], [2,2])
```

We can use these channels to define new states and predicates once we have described state transformation $\gg$ and predicate transformation $\ll$.

3.4.2 *State transformation*

Diagram (3.7) describes how a state s with the same type as the domain of a channel c gives rise to a state $c \gg s$ on the codomain of the channel. We sometimes say: 'state s is pushed forward along the channel c'.

How this works is best described for a channel $c\colon [n] \to [m]$, as represented in (3.5). A state s, corresponding to an $n \times n$ matrix S, gives an $m \times m$ matrix $c \gg s$ with at position (ℓ, k) the number:

(3.9) $\quad \mathrm{tr}(c_{k\ell} \cdot S) \quad$ where $\quad c_{k\ell}$ is the $n \times n$ matrix at (k, ℓ) in (3.5).

Notice the change of order of indices: at position (ℓ, k) of $c \gg s$ we use the matrix $c_{k\ell}$. What we use at each position is the Hilbert-Schmidt inner product, given by $\langle A \mid B \rangle = \mathrm{tr}(A^* \cdot B)$.

The same mechanism works for more complicated types $[n_1, \ldots, n_a]$.

We can now re-define the states `plus` and `minus` from the beginning of Section 3.1 as:

```
>>> plus = hadamard >> ket(0)
>>> minus = hadamard >> ket(1)
>>> plus
[[ 0.5+0.j  0.5+0.j]
 [ 0.5+0.j  0.5+0.j]]
>>> minus
[[ 0.5+0.j -0.5+0.j]
 [-0.5+0.j  0.5+0.j]]
```

We also see the familar equations: `cnot >> ket(i,j)` equals `ket(i,i+j)` where `+` is binary addition.

```
>>> cnot >> ket(0,0)
[[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]]
>>> cnot >> ket(0,1)
[[ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]]
>>> cnot >> ket(1,0)
[[ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j]]
>>> cnot >> ket(1,1)
[[ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]]
```

The action x/y/z channels on qubits in their Bloch sphere form can be described via the following tests.

```
>>> t = random.uniform(0, pi)
>>> p = random.uniform(0, 2*pi)
>>> x_chan >> bloch_state(t, p) == bloch_state(pi - t, 2*pi - p)
True
>>> y_chan >> bloch_state(t, p) == bloch_state(pi - t, pi - p)
True
>>> z_chan >> bloch_state(t, p) == bloch_state(t, pi + p)
True
```

3.4.3 *Predicate transformation*

Where state transformation works in the forward direction of a channel, predicate transformation acts in the backward direction: it transforms a predicate p with the same type as the codomain of a channel c into a predicate $c \ll p$ with the same type as the domain of the channel, see Diagram (3.7). We often say: 'predicate p is pulled back along channel c'. The resulting predicate $c \ll p$ can be understood as the weakest precondition[19] of p along c.

We again describe its mathematical definition for a channel $c\colon [n] \to [m]$ with types of length 1. Let predicate p have $m \times m$ matrix P. We construct an $n \times n$ matrix $c \ll p$ as sum of scaled matrices:

$$\sum_{k,\ell} P_{k\ell} \cdot c_{k\ell} \tag{3.10}$$

If we view a channel $[n] \to [m]$ as a map $c\colon \mathsf{M}_m \to \mathsf{M}_n$ between von Neumann algebras of matrices — in the opposite direction! — then c is determined by its action on base vectors, as $c_{k\ell} = c(E_{k\ell}) = c(|k\rangle\langle \ell\,|)$. Then $c(P) = \sum_{k,\ell} P_{k\ell} \cdot c_{k\ell}$ is the action of c on P, obtained by linear extension.

As illustration, the predicates `A2`, `B2` in the Bell table Example 23 can now be described via predicate transformation as:[20]

```
>>> A1 = hadamard << point_pred(0,2)
>>> A1
[[ 0.5+0.j  0.5+0.j]
 [ 0.5+0.j  0.5+0.j]]
>>> angle = pi / 3
>>> A2 = phase_shift(angle) << A1
>>> A2
[[ 0.50+0.j         0.25+0.4330127j]
 [ 0.25-0.4330127j  0.50+0.j       ]]
```

We can also *test* the transformation validity equation (3.8), even in an iterated manner, using compositon $*$ of channels:

```
>>> p = random_pred([2])
>>> s = random_state([2])
>>> x_chan >> (hadamard >> s) >= p
0.66686812144481133
>>> (x_chan * hadamard) >> s >= p
0.66686812144481133
>>> s >= hadamard << (x_chan << p)
0.66686812144481133
>>> s >= (x_chan * hadamard) << p
0.66686812144481133
```

Notice the reversal of channels when using $\ll$ twice, because $\ll$ works in backward direction.

[19] E. D'Hondt and P. Panangaden. Quantum weakest preconditions. *Math. Struct. in Comp. Sci.*, 16(3):429–451, 2006; and M. Ying. *Foundations of Quantum Programming*. Morgan Kaufmann, 2016

[20] See again the the reference `https://arxiv.org/abs/1406.7386` from sidenote 9 for details.

We have seen how states and predicates from discrete classical probability can be incorporated in the quantum world by putting the relevant probabilities at the diagonal. The next example illustrates how channels from discrete probability fit in the quantum picture.

Example 27 *Consider the channel* $c\colon\{0,1\} \to \{0,1,2\}$ *in discrete probability, see Section 1.4, given by:*

$$c(0) = \tfrac{1}{2}|0\rangle + \tfrac{1}{8}|1\rangle + \tfrac{3}{8}|2\rangle \qquad \text{and} \qquad c(1) = \tfrac{1}{3}|0\rangle + \tfrac{1}{2}|1\rangle + \tfrac{1}{6}|2\rangle.$$

We turn it into a quantum channel $[2] \to [3]$ *via the the follow* 3×3 *matrix of* 2×2 *matrices.*

$$M = \begin{pmatrix} \begin{pmatrix} 1/2 & 0 \\ 0 & 1/3 \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 1/8 & 0 \\ 0 & 1/2 \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 3/8 & 0 \\ 0 & 1/6 \end{pmatrix} \end{pmatrix}$$

We see how the probabilities from the discrete channel c *are organised on the diagonals of the matrices on the diagonal.*

For a state $\omega = \frac{1}{5}|0\rangle + \frac{4}{5}|1\rangle$ *on* 2 *we obtain the transformed state* $c_*(\omega) = \frac{11}{30}|0\rangle + \frac{17}{40}|1\rangle + \frac{5}{24}|2\rangle$ *on* 3*, via the formula (1.5). When we incorporate the above matrix* M *into an EfProb quantum channel* `d` *we get the same outcomes, on the diagonal, via quantum state transformation:*

```
>>> d >> probabilistic_state(1/5, 4/5)
[[ 0.36666667+0.j  0.00000000+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.42500000+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.00000000+0.j  0.20833333+0.j]]
```

Similarly for predicate transformation. Take the predicate q *on* 3 *defined by* $q(0) = \frac{1}{2}$, $q(1) = 0$, $q(2) = 1$*, then* $c^*(q)(0) = \frac{5}{8}$ *and* $c^*(q)(1) = \frac{1}{3}$ *using (1.6). These values again appear on the diagonal if we do quantum predicate transformation:*

```
>>> d << probabilistic_pred(1/2, 0, 1)
[[ 0.62500000+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.33333333+0.j]]
```

3.4.4 *Sequential and parallel composition of channels*

As we have already discussed at a mathematical level in the beginning of this section, channels can be composed sequentially: `d * c` is the channel obtained by first doing `c` and then `d`. Thus, the sequential composition operation $*$ is like function composition $\circ$ in mathematics, and is best pronounced as 'after'. Sequential composition `d * c` is well-defined if the codomain/output type of `c` is the same as the domain/input type of `d`; otherwise a Python exception is thrown.

Sequential composition of channels interacts with state- and predicate- transformation in the 'obvious' way: the left- and right hand side below are the same:

```
(d * c) >> s        is        d >> (c >> s)
(d * c) << p        is        c << (d << p)
```

Notice the reversal in order of the channels `c` and `d` in predicate transformation. The reason is that predicate transformation works in a backward direction.

In addition, there are the standard equalities form category theory, saying that identity channels `idn`, see Subsection 3.4.5 are identities for sequential composition `*`, and that composition is associative.

Parallel composition `c @ d` is always defined: there are no type restrictions on `c` and `d`, as illustrated in (3.6). This parallel composition `@` for channels interacts appropriately with composition `@` for states and with `@` for predicates, in the sense that:

```
(c @ d) >> (s @ t)        is        (c >> s) @ (d >> t)
(c @ d) << (p @ q)        is        (c << p) @ (d << q)
```

The 'interchange law' for monoidal categories also holds:

```
(c @ d) * (e @ f)        is        (c * e) @ (d * f)
```

Finally, channels are also closed under convex sums, via a function `convex_channel_sum`, which works just like the function `convex_state_sum` discussed in Subsection 3.1.1.

3.4.5 *Structural channels*

This subsection describes several 'structural' channels, namely identity, discard, and projection, swap, Kronecker, and ancilla. They will be described one by one. We occasionally use quantum circuits to visually explain what these channels do.[21]

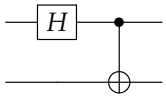
Identity channels For each domain `dom` (list of dimensions) there is an identity channel `dom -> dom` written as `idn(dom)`. This identity channel has no effect in sequential composition `*` of channels, but it is extremely useful in parallel composition `@`, for instance in defining the Bell states from the end of Section 3.1 more systematically via the Bell channel as:

```
bell_chan = cnot * (hadamard @ idn([2]))
bell00 = bell_chan >> ket(0,0)
bell01 = bell_chan >> ket(0,1)
bell10 = bell_chan >> ket(1,0)
bell11 = bell_chan >> ket(1,1)
```

[21] Wires in quantum circuits are meant to transfer single qubits. They have dimension 2 in the setting of the *EfProb* library. A purist might complain that we are slightly abusing the circuit notation, since we use it for channels and not for gates, as is traditional. To be more concrete, tranditionally one would write —|U|— for a unitary matrix U, used as *gate*, where we use —|c|— where c is a *channel*. Of course, this channel result from applying the function `channel_from_unitary` to the gate `U`, so that both cases amount to the same.

We can draw the channel `c` as a circuit:

```
bell_chan  =  cnot ∗ (hadamard @ idn([2]))  =
```



Unfortunately, we have to deal with a clash of conventions: flows in circuits go from left to right, whereas function composition works in the other direction, since ∗ is read as 'after'.

But the important thing to note is that the identity channel `idn([2])` in the definition of `bell_chan` is the wire in parallel with the Hadamard channel H, written below H in the above circuit.

One of the four bell states defined above, `bell00`, was already used in Example 23 to obtain the probabilities in the Bell table.

Example 28 *At the end of Section 3.1 we have introduced the eight Greenberger-Horne-Zeilinger (GHZ) states concretely via matrices. At this stage we can give a more abstract description, using state transformation via various channels:*

```
ghz = (idn([2]) @ cnot) >> ((bell_chan @ idn([2])) >> ket(0,0,0))
```

This state is called `ghz1` *in Section 3.1:*[22]

```
>>> ghz == ghz1
True
```

[22] Here we use == for equality of states. It is pre-defined and involves equality of domains, and equality of density matrices, up to a certain small value (the 'tolerance'), as incorporated in `numpy`'s `isclose` function. There is a similar equality function for predicates.

In *EfProb* an identity channel can work on an arbitary domain, given as list of dimentions. In circuits, wires represent qubit channels, of dimension 2. Hence three qubit wires in parallel can be described as:

```
idn([2,2,2])  =  ———  =  idn([2]) @ idn([2]) @ idn([2])
                 ———
                 ———
```

Discard channels Recall that in quantum theory we can discard — but not copy — resources. That's why we can marginalise states. For these same reasons we have channels for discarding, and for projection, see below.

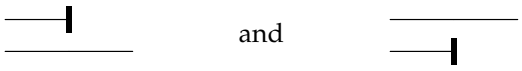
The discard channel has type `dom -> []` for an arbitrary list of dimensions `dom`. It simply terminates everything coming through. A discard channel `[2] -> []` is drawn as:

```
discard([2])  =  ——|
```

The outcome of a discard channel is independent of the input, see for instance:

```
>>> discard([2]) >> ket(0)
[[ 1.+0.j]]
>>> discard([2]) >> ket(1)
[[ 1.+0.j]]
```

Projection channels Now that we have discard channels, we can form projection channels, namely via a discard in parallel with an identity channel, as suggested in:



and

Here is a simple test:

```
>>> s = random_state([2])
>>> s
[[ 0.47726159+0.j        -0.44585810-0.0353013j]
 [-0.44585810+0.0353013j  0.52273841+0.j       ]]
>>> t = random_state([2])
>>> ((idn([2]) @ discard([2])) * cnot) >> (s @ t)
[[ 0.47726159+0.j          0.21691432+0.01717443j]
 [ 0.21691432-0.01717443j  0.52273841+0.j        ]]
>>> (cnot >> (s @ t)) % [1,0]
[[ 0.47726159+0.j          0.21691432+0.01717443j]
 [ 0.21691432-0.01717443j  0.52273841+0.j        ]]
```

These last lines illustrate that marginalisation coincides with projection.

In the same way, weakening corresponds to predicate transformation with a projection channel.

Swapping Sometimes it is convenient when wires can cross. There is a `swap` channel that can do this, but only for qubits. Its type is `[2,2] -> [2,2]`. Here are some simple examples.

```
>>> s = random_state([2])
>>> t = random_state([2])
>>> swap >> (s @ t) == t @ s
True
>>> swap >> bell00 == bell00
True
```

The other Bell states `bell01` *etc.* are also 'symmetric'.

Here is where swapping is useful. In the *EfProb* library the control bit in the `cnot` channel is the first (upper) argument, as depicted in:

`cnot` =

If you want "`cnot` upside-down" you can do this via swapping both the input and the output:

`swap * cnot * swap` =

There is no pre-defined swap channel for other dimensions.

Kronecker channels The product `@` of two states of type `[n]` and `[m]` is of type `[n,m]`. The size of the underlying density matrix is `n*m`, since products are implemented via the Kronecker product of matrices, see Subsection 3.1.1. The same works for parallel conjunctions `@` of predicates and for products `@` of channels.

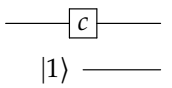
It turns out to be convenient to have channels that translate between `[n,m]` and `n*m`. This is done by what we call Kronecker channels, written as

```
kron : [n,m] -> [n*m]    and    kron_inv : [n*m] -> [n,m]
```

Internally these channels do nothing — they are just 'identity channels' — but they re-order inputs and outputs. They will be useful when we do measurements in bases of length greater than two, see the GHZ teleportation and superdense coding illustrations in Examples 31 and 32.

With these Kronecker channels we step outside the world of quantum circuits where wires correspond to quantum (or classical) bits. Indeed, we can now combine such qubit wires into ququat wires, via `[2,2] -> [4]`, or decompose them via first applying the Kronecker-inverse `[4] -> [2,2]` and then using projection/discard channels.

Ancilla channels Channels can 'narrow' or 'widen' over time: they can narrow via discarding certain flows. But new flows can also be added to a channel, via new states, so that a channel widens. Such new states are often called *ancilla* qubits. Pictorially, the can be added to an existing channel c in the following way.

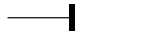



```
= c @ ket(1).as_chan()
```

Here we use that each state of type `dom` has a method `as_chan` which turns in to a channel `[] -> dom`. By using this ancilla state/channel in parallel with an existing channel, this existing channel is widenend. This effect is illustrated in:

```
>>> c = hadamard
>>> s = random_state([2])
>>> (c @ ket(1).as_chan()) >> s == (c >> s) @ ket(1) )
True
```

The combination of discard and ancilla makes it possible to terminate a wire and restart it with a new qubit on it, as in:




```
= discard([2]) @ ket(0).as_chan()
= ket(0).as_chan() @ discard([2])
```

Indeed, the random state `s` plays no role in this construction:

```
>>> s = random_state([2])
>>> (discard([2]) @ ket(0).as_chan()) >> s
[[ 1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j]]
>>> (ket(0).as_chan() @ discard([2])) >> s
[[ 1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j]]
```

An arbitrary state `w` can be reconstructed from the associated channel `w.as_chan()` of type `[] -> dom`, namely by transforming the trivial state `init_state` of type `[]` via this channel. Similarly, predicate transformation with `w.as_chan()` corresponds to validity in state `w`.

```
>>> dom = [2,5]
>>> w = random_state(dom)
>>> w == w.as_chan() >> init_state
True
>>> p = random_pred(dom)
>>> w >= p
0.273592934749
>>> w.as_chan() << p
[[ 0.27359293 -2.38524478e-18j]]
```

We see that essentially the same probability `w >= p` appears via predicate transformation, as a trivial matrix, with an almost-zero imaginary part arising from rounding errors.

3.4.6 Measurement, classical control, and instruments

This subsection introduces measurement, both for predicates and for tests. It also describes how to use the outcomes of such measurements in classical control, where only the probabilistic outcomes of the measurement are used. Multiple examples are given in Section 3.5, of teleportation and superdense coding, both using shared Bell and GHZ states.

Measurement For each predicate `p` of type `dom` there is a channel `meas_pred(p) : dom -> [2]`. It is such that for an arbitrary state `s`,

$$\texttt{meas\_pred(p) >> s} \;=\; \begin{pmatrix} r & 0 \\ 0 & 1-r \end{pmatrix} \qquad \text{where} \qquad r \;=\; \texttt{s >= p}.$$

Hence this channel `meas(p)` measures and produces the resulting value as a probabilistic state. Concretely:

```
>>> p = random_pred([5])
>>> s = random_state([5])
>>> s >= p
0.514240393407
>>> meas_pred(p) >> s
```

```
[[ 0.51424039+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.48575961+0.j]]
```

The predicate `p` can be recovered as `meas_pred(p) << yes_pred`, and its orthosupplement `~p` as `meas_pred(p) << no_pred`.

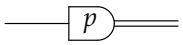
The following abbreviations are convenient for binary measurement in the standard basis:

```
meas0 = meas_pred(point_pred(0,2))
meas1 = meas_pred(point_pred(1,2))
```

For instance, a (fair) quantum coin can be obtained as:

```
>>> meas0 >> (hadamard >> ket(0))
[[ 0.5+0.j  0.0+0.j]
 [ 0.0+0.j  0.5+0.j]]
>>> (meas0 * hadamard) >> ket(1)
[[ 0.5+0.j  0.0+0.j]
 [ 0.0+0.j  0.5+0.j]]
```

In quantum circuits it is custom to indicate the classical outcome of such a measurement by using a double wire ═══. Quantum wires are indicated by single lines ———. For a measurement channel `meas(p)` we use a picture:



Where for the above two special case we simply write:

`meas0` = 

`meas1` = 

These measurement ideas are used in a channel `classic(dom) : dom -> dom` that measures in the 'computational basic' of the domain. This channel removes everything from a state except its diagonal:

```
>>> s = random_state([3])
>>> s
[[ 0.32345901+0.j          0.00687158+0.15788094j  0.15185772-0.22324826j]
 [ 0.00687158-0.15788094j  0.34920366+0.j         -0.23730879+0.01392982j]
 [ 0.15185772+0.22324826j -0.23730879-0.01392982j  0.32733733+0.j
]]
>>> classic([3]) >> s
[[ 0.32345901+0.j  0.00000000+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.34920366+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.00000000+0.j  0.32733733+0.j]]
```

We recall from Subsection 1.2.1 that a test is a list of predicates whose joint sum + exists and equals the truth predicate. These tests can also be used for measurement, generalising what is commonly called 'measurement in a basis'. Here is a Bell measurement illustration, using a measurement channel `meas_test` that uses a test as input.

```
>>> bell_test = [bell00.as_pred(), bell01.as_pred(), bell10.as_pred(), bell11.as_pred()]
>>> meas_bell = meas_test(bell_test)
>>> w = cnot >> random_state([2,2]
>>> meas_bell >> w
[[ 0.12947764+0.j  0.00000000+0.j  0.00000000+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.16059948+0.j  0.00000000+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.00000000+0.j  0.31687829+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.00000000+0.j  0.00000000+0.j  0.39304459+0.j]]
>>> w >= bell00.as_pred()
0.129477644519
>>> w >= bell01.as_pred()
0.160599477211
>>> w >= bell10.as_pred()
0.316878289196
>>> w >= bell11.as_pred()
0.393044589075
```

We see that measurement of an (arbitrary) state `w : [2,2]` in the Bell basis, turned into a test of predicates, yields a probabilistic state. These probabilities on the diagonal can be obtained separately as validities of the individual predicates in the state `w`.

Control channels In Subsection 3.4.1 we have seen the `cnot` channel, obtained from a unitary matrix. It performs what is called 'controlled-not'. One would like to have similar 'controlled' channels. At this stage we only support 'classical control'. The more adventurous 'quantum control'[23] also known as 'alternation'[24] is not yet supported in *EfProb*.

[23] M. Ying. *Foundations of Quantum Programming*. Morgan Kaufmann, 2016

[24] C. Bădescu and P. Panangaden. Quantum alternation: Prospects and problems. QPL 2016, 2016

For a channel `c : dom -> cod` there is a classical control channel `ccontrol(c) : [2]+dom -> [2]+cod` with an additional qubit input. If this qubit is `ket(0)`, the control channel is the identity on the second input. Otherwise it applies the channel `c`. This behaviour is illustrated in:

```
>>> s = random_state([2])
>>> s
[[ 0.58141072+0.j         0.46816135+0.1293966j]
 [ 0.46816135-0.1293966j  0.41858928+0.j       ]]
>>> (ccontrol(hadamard) >> (ket(0) @ s)) % [0,1]
[[ 0.58141072+0.j         0.46816135+0.1293966j]
 [ 0.46816135-0.1293966j  0.41858928+0.j       ]]
>>> hadamard >> s
[[ 0.96816135+0.j         0.08141072-0.1293966j]
 [ 0.08141072+0.1293966j  0.03183865+0.j       ]]
>>> (ccontrol(hadamard) >> (ket(1) @ s)) % [0,1]
[[ 0.96816135+0.j         0.08141072-0.1293966j]
```

The `cnot` channel defined in Subsection 3.4.1 is *not* defined via classical control, but via quantum control. If we restrict `cnot` to classical

bits via the `classic` channel we can make a precise statement, namely that the next two channels are the same.

- `cnot * (classic([2]) @ idn([2]))`
- `ccontrol(x_chan)`

There is a more general n-ary version of `ccontrol`, which we call `ccase`, for 'classical case'. It takes a list of channels as input. These channels all need to have the same domain and codomain, say `dom` and `cod`. If the list of channels has length `n`, then the `ccase` channel has type `[n]+dom -> [n]+cod`. The main property of `ccase` is illustrated below.

```
>>> s = probabilistic_state(0.2, 0.3, 0.5)
>>> t = random_state([2])
>>> w = ccase(x_chan, hadamard, idn([2])) >> (s @ t)
>>> w % [1, 0]
[[ 0.2+0.j  0.0+0.j  0.0+0.j]
 [ 0.0+0.j  0.3+0.j  0.0+0.j]
 [ 0.0+0.j  0.0+0.j  0.5+0.j]]
>>> w % [0, 1]
[[ 0.62046813+0.j  0.22676644+0.j]
 [ 0.22676644+0.j  0.37953187+0.j]]
>>> convex_state_sum( (0.2, x_chan >> t), \
...                  (0.3, hadamard >> t), \
...                  (0.5, idn([2]) >> t) )
[[ 0.62046813 +0.00000000e+00j  0.22676644 -2.77555756e-17j]
 [ 0.22676644 +2.77555756e-17j  0.37953187 +0.00000000e+00j]]
```

The `ccase` channel returns the probabilistic state `s` in its first component. Its second output component is a convex combination of the channels applied to the second input component.

Instruments As described above, measurement destroys the state and only returns the probabilities of the measurement, as classical state. Instruments, as introduced in[25], combine measurement and state update/revision, by producing not only probabilities but also a convex sum of conditional states. Instruments can be defined for tests, but *EfProb* currently only supports them for predicates.

Let `p` be a predicate with domain `dom`. The channel `instr(p)` has type `dom -> [2]+dom`. By projecting away the second part we obtain measurement:

[25] B. Jacobs. New directions in categorical logic, for classical, probabilistic and quantum logic. *Logical Methods in Comp. Sci.*, 11(3):1–76, 2015

```
>>> p = random_pred([5])
>>> (idn([2]) @ discard([5])) * instr(p) == meas_pred(p)
True
```

The second part of the output of an instrument is a convex combination of conditional states, as illustrated in:

```
>>> p = random_pred([5])
>>> s = random_state([5])
>>> discard([2]) @ idn([5]) * instr(p) >> s ==
...   convex_state_sum( (s >= p, s / p), (s >= ~p, s / ~p)
True
```

The quantum circuit language of Quipper[26] involves only destructive measurement, via what we call the `meas_pred` channel. One of the saillant features of effectus theory is that it explicitly deals with side-effects (state changes) of measurements. This is incorporated in the instrument channel `instr`, via conditioning of states as described above.

[26] A. Green., P. LeFanu Lumsdaine, N. Ross, P. Selinger, and B. Valiron. Quipper: A scalable quantum programming language. In *Proc. 34th ACM SIGPLAN Conf. on Progr. Language Design and Implementation*, pages 333–342. ACM, 2013

Conditioning of states `s/p` is intimately linked to sequential conjunction `p & q` of predicates, via Bayes' rule, see Subsection 3.2.3. This connection re-appears when we describe predicate transformation for instruments, as described below. We have to weaken the predicate `q` so that domains fit.

```
>>> p = random_pred([10])
>>> q = random_pred([10])
>>> instr(p) << truth([2]) @ q == (p & q) + (~p & q)
True
>>> instr(p) << point_pred(0,2) @ q == p & q
True
>>> instr(p) << point_pred(1,2) @ q == ~p & q
True
```

We see that by using `unit_pred(2,i)`, which is the same as `ket(i).as_pred()`, we get the sequential conjunctions separately.

We take this instrument channel one step further and incorporate it into an if-then-else style construction, written as `pcase(p)(c,d)`, for 'predicate-case'. This function `pcase` is applied to a predicate `p` and to a pair of channels `c,d`. Intuitively it performs if-`p`-then-`c`-else-`d`, but it does so by taking probabilities and side-effect into account.

```
>>> p = random_pred([2])
>>> s = random_state([2])
>>> pcase(p)(x_chan, y_chan) >> s ==
...   convex_state_sum( (s >= p, x_chan >> s/p), (s >= ~p, y_chan >> s/~p) )
True
>>> q = random_pred([2])
>>> pcase(p)(x_chan, y_chan) << q ==
...   (p & x_chan << q) + (~p & y_chan << q) )
True
```

The latter equality corresponds to the familar weakest precondition rule for if-then-else.

Finally, by conditioning the outcome of an instrument to the "true" case and taking the second marginal we can extract the revised

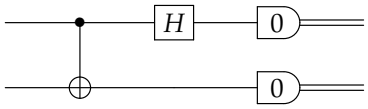
state:

```
>>> p = random_pred([10])
>>> s = random_state([10])
>>> ((instr(p) >> s) / (point_pred(0,2) @ truth([10]))) % [0,1] ==
s / p
True
```

3.5 *Teleportation and superdense coding examples*

This section only contains examples, namely of the famous teleportation and superdense coding protocols. Both these protocols are described in two versions, with the Bell state and with the GHZ state as shared state.

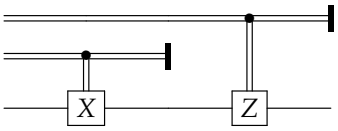
Example 29 *In the teleportation protocol a qubit is transferred via two classical bits and two entangled qubits. The two parties involved are called* `alice` *and* `bob`*. The quantum circuit for* `alice` *is:*



In EfProb this becomes:

```
alice = (meas0 @ meas0) * (hadamard @ idn([2])) * cnot
```

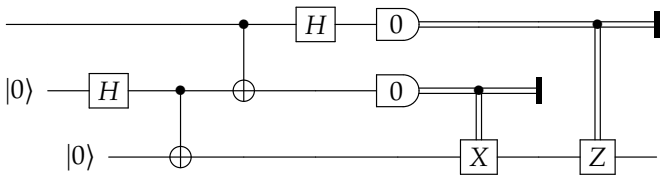
The circuit for `bob` *is:*



As a channel it is:

```
bob = (discard([2]) @ idn([2])) * ccontrol(z_chan)
          * (idn([2]) @ discard([2]) @ idn([2])) * (idn([2]) @ ccontrol(x_chan))
```

The circuit for the whole protocol first prepares the Bell state `bell00`*, followed by* `alice` *and then* `bob`*:*



As a channel it is:

```
teleportation = bob * (alice @ idn([2])) * (idn([2]) @ bell00.as_chan())
```

In this way, `teleportation` *is a channel* `[2] -> [2]`*. The claim is that this channel is the identity. The EfProb library does not provide a way to* prove *this logically. But it does allow us to* test *the claim that* `teleportation` *is the identity channel, by applying it to an arbitrary state. After loading the above definitions we can do this:*

```
>>> s = random_state([2])
>>> s
[[ 0.80038663+0.j          0.01459339+0.11782929j]
 [ 0.01459339-0.11782929j  0.19961337+0.j        ]]
>>> teleportation >> s
[[ 0.80038663+0.j          0.01459339+0.11782929j]
 [ 0.01459339-0.11782929j  0.19961337+0.j        ]]
```

Of course, testing does not prove anything. But it can disprove *(falsify) and thus reveal errors. More succintly, the fact that the original state* `s` *re-appears after teleportation can be tested as:*

```
>>> s == (teleportation >> s)
True
```

We can actually do a little bit more. If we claim that the `teleportation` *and* `idn([2])` *channels are the same, we can check that they have the same matrices, via their* `array` *attribute. Checking actual equality does not work, because of the small inaccuracies in floating point calculations. By Python's matrix library* `numpy`*, commonly abbreviated (imported) as* `np`*, provides a function* `isclose` *that tells if all the numbers in the two argument matrices are really close.*

```
>>> np.isclose(teleportation.array, idn([2]).array)
[[[[ True  True]
   [ True  True]]

  [[ True  True]
   [ True  True]]]


 [[[ True  True]
   [ True  True]]

  [[ True  True]
   [ True  True]]]]
```

This looks good! We can reduce the answer to one boolean by adding the `numpy` *function* `all`*, as in:*

The combination of `all` and `isclose` exists in `numpy` as `allclose`.

```
>>> np.all(np.isclose(teleportation.array, idn([2]).array))
True
```

We could have saved ourselves this trouble, via the pre-defined equality of channels, which checks equality of domains and codomains, and 'closedness' of arrays, as described above, see also Sidenote 22*.*

```
>>> teleportation == idn([2])
True
```

Again, we emphasise: this is not a proof, at most an indication.

This last equality of channels looks pleasingly simple. But it involves non-trivial computation: the intermediate channels, after adding the ancilla Bell state and before projecting, have type `[2,2,2] -> [2,2,2]`*. They involve an* 8×8 *matrix of* 8×8 *matrices, and thus a* 64×64 *matrix.*

Example 30 *The superdense coding protocol can be seen as a reverse version of teleportation: it allows to transfer two classical bits via an entangled quantum state (again a Bell state). The relevant channels are:*

```
alice = (discard([2]) @ idn([2])) * ccontrol(x_chan) * (idn([2]) @ proj2)
            * (idn([2]) @ ccontrol(z_chan)) * (swap @ idn([2]))
  bob = (meas0 @ meas0) * (hadamard @ idn([2])) * cnot
```

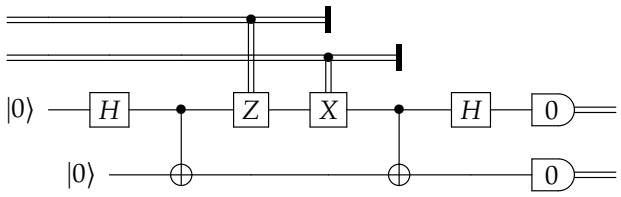
These two channels are combined with a Bell state in a function that takes two probabilities `r,s` *as input:*

```
def superdense_coding(r, s):
    return bob >> ((alice @ idn([2])) >> (cflip(r) @ cflip(s) @ bell00))
```

We can test this function by feeding it with arbitrary random numbers from $[0,1]$*, using Python's* `random` *module.*

```
>>> r = random.uniform(0,1)
>>> s = random.uniform(0,1)
>>> r
0.28103908384848075
>>> s
0.08831266497462498
>>> superdense_coding(r,s) % [1,0]
[[ 0.28103908+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.71896092+0.j]]
>>> superdense_coding(r,s) % [0,1]
[[ 0.08831266+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.91168734+0.j]]
```

For those who prefer quantum circuits we include the whole protocol as such:



In the end one may wander if it is possible to obtain an equality of channels for superdense coding, like at the end of Example 29 for teleportation.

What is different here is that the superdense coding protocol acts on classical *(probabililistic) bits, so that we will never obtain that its channel is equal to the identity channel. What we can hope for is an equality with the classic channel, which extracts the probabilistic part from qubits via measurement in the standard basis.*

```
>>> sdc = bob * (alice @ idn([2])) * (idn([2,2]) @ bell00.as_chan())
>>> sdc.dom
[2,2]
>>> sdc.cod
[2,2]
>>> sdc == classic([2,2])
True
```

We include two more examples which essentially achieve the same, but now use shared GHZ states instead of Bell states for teleportation and superdense coding. These examples are interesting since they involve a more general form of measurement, not wrt. a predicate, but wrt. a test.

Example 31 *In the teleportation protocol with a shared GHZ state Alice has an unknown state* `s` *that she wishes to transport to Bob. Alice and Bob share a GHZ state, where Alice has access to the first qubit of this tripartite state* `ghz`*, and Bob has access to the second and third qubit.*[27] *Alice jointly measures the state-to-be-transported* `s` *and her part of the state* `ghz` *in the Bell basis, given by the four states* `bell00`*,* `bell01`*,* `bell10` *and* `bell11`*. Because measurement in this basis yields 4 outcomes, instead of 2, we can no longer use quantum circuits. Therefore we shall be more explicit about the types (domains and codomains) of the various channels.*

On Alice's side we use the following set-up.

[27] In a variation on this protocol Bob has access to the second qubit of `ghz` and a third participant Charlie has access to the third qubit. In this set-up one can achieve 'secret sharing', where Alice shares an unknown quantum state, and Bob and Charlie have to cooperate to obtain that state, see *e.g.* .

M. Hillery, V. Buzek, and A. Berthiaume. Quantum secret sharing. *Phys. Rev. A*, 59:1829–1834, 1999

```
>>> bell_test = [bell00.as_pred(), bell01.as_pred(), bell10.as_pred(), bell11.as_pred()]
>>> meas_bell = meas_test(bell_test)
>>> meas_bell.dom
[2,2]
>>> meas_bell.cod
[4]
>>> alice = (meas_bell @ idn([2,2])) * (idn([2]) @ ghz.as_chan())
>>> alice.dom
[2]
>>> alice.cod
[4,2,2]
```

As we see, this measurement channel yields a (classic/probabilistic) state of dimension 4, on which case distinctions can be made by Bob.

But first Bob measures his qubit in the Hadamard basis, given by the states `plus` *and* `minus`*. Subsequently we combines the resulting 2 options with the 4 options obtained from Alice, via the Kronecker channel* `kron`*, into*

a joint probabilistic state with 8 options. Then he can make the relevant case distinctions:

```
>>> hadamard_test = [plus.as_pred(), minus.as_pred()]
>>> meas_hadamard = meas_test(hadamard_test)
>>> bob = ( discard([8]) @ idn([2]) ) \
...       * ccase(idn([2]), z_chan, x_chan, x_chan * z_chan, \
...               z_chan, idn([2]), x_chan * z_chan, x_chan) \
...     * ( kron(4, 2) @ idn([2]) ) \
...     * ( idn([4]) @ meas_hadamard @ idn([2]) )
>>> bob.dom
[4, 2, 2]
>>> bob.cod
[2]
>>> ghz_teleporation = bob * alice
```

Since the types match, the latter composite is allowed, giving us GHZ-teleportation as a single channel. Now we can test it:

```
>>> s = random_state([2])
>>> s
[[ 0.33869735+0.j          0.06518770+0.17403915j]
 [ 0.06518770-0.17403915j  0.66130265+0.j        ]]
>>> ghz_teleporation >> s
[[ 0.33869735+0.j          0.06518770+0.17403915j]
 [ 0.06518770-0.17403915j  0.66130265+0.j        ]]
>>> ghz_teleporation == idn([2])
True
```

Example 32 *Superdense coding can also be done with GHZ as shared state*[28]*. In the example bewlow three classical (probabilistic) qubits are transferred via two ordinary qubits. These three classical qubits give rise to 8 options for a controlled case channel. The state* `ghz` *is used as shared state. Alice has access to the first two qubits of* `ghz` *and applies a controlled case channel to them. One new channel is needed, which is commonly called* `iy`*. It is defined first.*

[28] J. Cereceda. Quantum dense coding using three qubits. See arxiv.org/abs/quant-ph/0105096, 2001; and A. Hillebrand. Superdense coding with GHZ and quantum key distribution with W in the ZX-calculus. See arxiv.org/abs/1210.0650, 2012

```
>>> iy_matrix = np.array([[0,1],
...                       [-1,0]])
>>> iy_chan = channel_from_isometry(iy_matrix, [2], [2])
>>> alice = (discard([8]) @ idn([2,2])) * ccase(idn([2]) @ idn([2]),
...                                             idn([2]) @ x_chan,
...                                             x_chan @ idn([2]),
...                                             x_chan @ x_chan,
...                                             z_chan @ idn([2]),
...                                             z_chan @ x_chan,
...                                             iy_chan @ idn([2]),
...                                             iy_chan @ x_chan)
>>> alice.dom
[8, 2, 2]
>>> alice.cod
```

```
[2, 2]
```

The role of Bob is simple: he measures the two incoming qubits and his own third qubit of the GHZ state in the GHZ basis.

```
>>> ghz_test = [ghz1.as_pred(), ghz2.as_pred(), ghz3.as_pred(), ghz4.as_pred(),
...             ghz5.as_pred(), ghz6.as_pred(), ghz7.as_pred(), ghz8.as_pred()]
>>> meas_ghz = meas_test(ghz_test)
>>> bob = meas_ghz
>>  ghz_sdc = bob * (alice @ idn([2])) * (idn([8]) @ ghz.as_chan())
>>> ghz_sdc.dom
[8]
>>> ghz_sdc.cod
[8]
```

We are ready to test the channel `ghz_superdense_coding`*. One way of doing this by taking an arbitrary probabilistic state of length 8, and feeding it into the channel.*

```
>>> s = random_probabilistic_state([8])
>>> t = ghz_sdc >> s
```

A manual comparison of states `s` *and* `t` *shows that they contain the same probabilities . . . but in a different order.*

We can get a better handle on the situation by translating the input and output of length 8 into 3 inputs of length 2, via the Kronecker channels.

```
>>> k1 = kron(4,2) * (kron(2,2) @ idn([2]))
>>> k2 = (kron_inv(2,2) @ idn([2])) * kron_inv(4,2)
>>> r1 = random_probabilistic_state([2])
>>> r2 = random_probabilistic_state([2])
>>> r3 = random_probabilistic_state([2])
>>> r1
[[ 0.04748214  0.        ]
 [ 0.          0.95251786]]
>>> r2
[[ 0.67005383  0.        ]
 [ 0.          0.32994617]]
>>> r3
[[ 0.43412549  0.        ]
 [ 0.          0.56587451]]
>>> r = k1 >> (r1 @ r2 @ r3)
>>> w = k2 >> (ghz_sdc >> r)
>>> w % [1,0,0]
[[ 0.43412549+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.56587451+0.j]]
>>> w % [0,1,0]
[[ 0.67005383+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.32994617+0.j]]
>>> w % [0,0,1]
[[ 0.04748214+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.95251786+0.j]]
```

We now see that the transferred classical bits appear in reverse order.

In the end we point out that the transfer from Alice to Bob is of type `[8,2,2] -> [8,2,2]`. *This involves a* 1024×1024 *matrix. The computation time for this protocol is indeed noticeable, in the order of a second (on an average laptop). The EfProb formalism hides these implementation details.*

Bibliography

S. Abramsky. Contextual semantics: From quantum mechanics to logic, databases, constraints, and complexity. *EATCS Bulletin*, 113, 2014.

S. Abramsky and A. Brandenburger. The sheaf-theoretic structure of non-locality and contextuality. *New Journ. of Physics*, 13:113036, 2011.

C. Bădescu and P. Panangaden. Quantum alternation: Prospects and problems. QPL 2016, 2016.

J. Cereceda. Quantum dense coding using three qubits. See `arxiv.org/abs/quant-ph/0105096`, 2001.

K. Cho and B. Jacobs. The EfProb library for probabilistic calculations. In F. Bonchi and B. König, editors, *Conference on Algebra and Coalgebra in Computer Science (CALCO 2017)*, volume 72 of *LIPIcs*. Schloss Dagstuhl, 2017.

K. Cho, B. Jacobs, A. Westerbaan, and B. Westerbaan. An introduction to effectus theory. see `arxiv.org/abs/1512.05813`, 2015.

E. D'Hondt and P. Panangaden. Quantum weakest preconditions. *Math. Struct. in Comp. Sci.*, 16(3):429–451, 2006.

A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Physical Review*, 47(10):777–780, 1935.

T. Gehr, S. Misailovic, and M. Vechev. PSI: Exact symbolic inference for probabilistic programs. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification*, number 9779 in Lect. Notes Comp. Sci., pages 62–83. Springer, Berlin, 2016.

N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008.

A. Green., P. LeFanu Lumsdaine, N. Ross, P. Selinger, and B. Valiron. Quipper: A scalable quantum programming language. In *Proc. 34th ACM SIGPLAN Conf. on Progr. Language Design and Implementation*, pages 333–342. ACM, 2013.

A. Hillebrand. Superdense coding with GHZ and quantum key distribution with W in the ZX-calculus. See arxiv.org/abs/1210.0650, 2012.

M. Hillery, V. Buzek, and A. Berthiaume. Quantum secret sharing. *Phys. Rev. A*, 59:1829–1834, 1999.

B. Jacobs. New directions in categorical logic, for classical, probabilistic and quantum logic. *Logical Methods in Comp. Sci.*, 11(3):1–76, 2015.

B. Jacobs. Hyper normalisation and conditioning for discrete probability distributions. See arxiv.org/abs/1607.02790, 2016.

B. Jacobs. A channel-based perspective on conjugate priors. See arxiv.org/abs/1707.00269, 2017a.

B. Jacobs. Quantum effect logic in cognition. *Journ. Math. Psychology*, 81:1–10, 2017b. See https://doi.org/10.1016/j.jmp.2017.08.004.

B. Jacobs. From probability monads to commutative effectuses. *Journ. of Logical and Algebraic Methods in Programming*, 156, 2017, to appear.

B. Jacobs and F. Zanasi. A predicate/state transformer semantics for Bayesian learning. In L. Birkedal, editor, *Math. Found. of Programming Semantics*, number 325 in Elect. Notes in Theor. Comp. Sci., pages 185–200. Elsevier, Amsterdam, 2016.

B. Jacobs and F. Zanasi. The logical essentials of Bayesian reasoning. See arxiv.org/abs/1804.01193, 2018.

B. Jacobs, B. Westerbaan, and A. Westerbaan. States of convex sets. In A. Pitts, editor, *Foundations of Software Science and Computation Structures*, number 9034 in Lect. Notes Comp. Sci., pages 87–101. Springer, Berlin, 2015.

A. McIver, C. Morgan, G. Smith, B. Espinoza, and L. Meinicke. Abstract channels and their robust information-leakage ordering. In M. Abadi and S. Kremer, editors, *Princ. of Security and Trust*, number 8414 in Lect. Notes Comp. Sci., pages 83–102. Springer, Berlin, 2014.

S. Michels, A. Hommersom, P. Lucas, and M. Velikova. A new probabilistic constraint logic programming language based on a generalised distribution semantics. *Artificial Intelligence*, 228:1–44, 2015.

B. Milch, B. Marthi, S. Russell, D. Sontag, D. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In L. Getoor and B. Taskar, editors, *Statistical Relational Learning*. MIT Press, 2007.

M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.

J. Paykin, R. Rand, and S. Zdancewic. QWIRE: A core quantum circuit language. POPL, 2017.

E. Rieffel and W. Polak. *Quantum Computing. A Gentle Introduction*. MIT Press, Cambridge, MA, 2011.

G. Schay. An algebra of conditional events. *Journ. Math. Analysis and Appl.*, 24:334–344, 1968.

D. Wecker and K. Svore. Liq*ui*|>: A software design architecture and domain-specific language for quantum computing, 2014. See arxiv.org/abs/1402.4467.

N.S. Yanovsky and M.A. Manucci. *Quantum Computing for Computer Scientists*. Cambridge Univ. Press, 2008.

M. Ying. *Foundations of Quantum Programming*. Morgan Kaufmann, 2016.

Index