FlatPPL, a Flat Portable Probabilistic Language

Expert-Level Proposal/Motivation and Design Draft

Oliver Schulz
Max Planck Institute for Physics, Garching/Munich, Germany
oschulz@mpp.mpg.de

Abstract FlatPPL is a declarative, inference-agnostic probabilistic language designed for authoring, sharing, and converting statistical models across scientific domains. It is intended both as a directly writable source language and as a portable representation that higher-level modeling frontends may emit. The design is still under development; this document presents the current draft. FlatPPL describes models as static directed acyclic graphs (DAGs) of named mathematical objects — variates, measures, functions, and likelihoods — in a single global namespace with no block structure, no loops, and no dynamic branching. Its canonical surface form is simple and lies within the intersection of Python and Julia syntax, making parsing lightweight and host-language embedding practical. In addition to deterministic and stochastic nodes, the language provides a measure algebra for measures and Markov kernels. Measures, kernels, and deterministic functions can be reified from sub-DAGs with optional boundary inputs, making it possible to extract conditional kernels and deterministic functions from larger models without auxiliary variables. FlatPPL defines profiles, subsets of the language that map to target languages like the HEP Statistics Serialization Standard (HS³). FlatPPL is accompanied by the Flat Probabilistic Intermediate Representation (FlatPIR), to facilitate term-rewriting for optimization and conversion between profiles.

Contents

Scope and status of this document.

This document is a design draft. It is intended for collaborators and technical experts. It is not a tutorial, user or reference manual, or a complete language specification.

The aim is to motivate FlatPPL and make the proposed semantics, syntax, and features concrete enough to discuss feasibility of implementation and interoperability, and to present concepts that might also be transferred to existing standards and frameworks. The document also aims to provide enough detail that domain scientists with a strong statistical background can evaluate how models from their disciplines would map to FlatPPL, identify abstractions and features still missing, and contribute to a design with wide scope.

This is a living document and the design is not frozen. Some details should be read as current proposals rather than final decisions. Readers are encouraged to test the language against realistic use cases and to treat areas where the draft falls short as useful feedback for the next iteration.

---

1 Context and motivation

1.1 Goals and target audience

Statistical modeling in the sciences requires tools that are both mathematically rigorous and practically durable. High Energy Physics (HEP) in particular has a decades-long tradition of rigorous statistical analysis, with code lifetimes measured in decades and a strong culture of reproducibility and model preservation. The HEP community and related fields — astrophysics, nuclear physics, and other data-intensive sciences — are a primary target audience for the modeling language proposed here, though FlatPPL is designed to be broadly applicable to statistical scientific models in general.

The goal is to create a common standard and infrastructure for serializing, sharing, and using statistical models — initially motivated by physics, but designed to be applicable across scientific fields. Models should be FAIR (Findable, Accessible, Interoperable, Reusable), with computational engines initially targeting C++, Python, and Julia. This document proposes FlatPPL — a declarative model description language — as a standalone specification for statistical models, designed with substantial compatibility with existing standards and tools (HS³, RooFit, pyhf).

This document serves both as a design proposal and as a language reference. New readers may want to read the first four sections (motivation, overview, value types, and language design), then consult the following reference-style chapters (measure algebra, functions, distributions) as needed. Later sections provide worked examples, interoperability guidance, and design rationale.

1.2 The starting point: RooFit and HS³

The current principal building blocks for statistical modeling in High Energy Physics are **RooFit** (a C++ modeling toolkit in ROOT) and the **HEP Statistics Serialization Standard (HS³)**, a JSON-based interchange format. **pyhf** is a pure-Python implementation of the HistFactory template-fitting subset of RooFit, with its own JSON serialization format.

These are great strengths to build on, but there are limitations as well. RooFit and HistFactory are tied to C++ and the ROOT framework. Their engine-independent serializations, HS³ and pyhf

JSON, are highly machine-parseable but also verbose and inconvenient for humans to write and review. FlatPPL is intended to offer wide scope with a concise canonical syntax, while maintaining clear bridges to these established standards.

RooFit provides a rich and mature framework for building probability models. Its architecture is based on directed acyclic graphs (DAGs) that express computational dependencies between named objects. These graphs support derived quantities, conditional products (`RooProdPdf` with `Conditional`), and marginalization (`createProjection`). However, stochastic dependencies — where one distribution's variate becomes another's parameter — require explicit conditional product construction; they are not inferred from the graph structure.

The concrete RooFit design, however, has some drawbacks, also in regard to formal clarity:

- **No distribution/PDF distinction.** RooFit conflates distributions with their PDFs, and PDFs do not separate parameters from observables — the distinction arises from usage context (which variables appear in the dataset at fit time). This allows operations such as normalizing a likelihood function over parameter space and treating it as a probability density — an operation that is statistically ill-defined in general, since the likelihood is not a probability measure on parameter space.
- **No vector-valued variables.** All variables are scalar `RooRealVar` objects — there are no vector-valued parameters or variates. Record-like structures (e.g. named components of a multivariate normal) must be flattened into individually named scalars in the global namespace.

HS³ defines a "forward-modelling" approach: a statistical model maps a parameter space to probability distributions describing experimental outcomes. It is a programming-language independent standard designed to be functionally compatible with RooFit but with clearer separation of some statistical concepts. HS³ is young, compared to RooFit, but already in use by the ATLAS collaboration for publishing likelihoods on HEPData.

HS³ has its own limitations:

- **No hierarchical stochastic composition.** HS³ supports parameter references and functional dependencies among named objects (a parameter of one distribution can be bound to the output of a function), but it doesn't yet provide a standard-level mechanism for hierarchical models. So while RooFit can express such models, it cannot serialize them to HS³ yet.
- **Scalar-only values.** Parameters, variates, and function outputs must all be scalar — only observed data may contain vectors, creating an asymmetry.
- **Readability.** JSON is machine-friendly but difficult for humans to write and review, particularly for complex models.

FlatPPL aims to combine RooFit's expressive power (hierarchical models, conditional products, measure algebra) with clean statistical semantics — in a form that can serve as an implementation-independent modeling language with substantial HS³ and RooFit compatibility. There is an active effort to evolve both HS³ and RooFit toward greater expressiveness; bidirectional compatibility with them, for a large class of models, is a design goal of FlatPPL.

1.3 Probabilistic languages

A probabilistic language is a formal language for declaring generative models — descriptions of how data could have been produced by a stochastic process. The literature partially distinguishes between probabilistic modeling languages and probabilistic programming languages, though the distinction is not always sharp. A probabilistic programming language is often understood to provide both model specification and automatic inference, though not all do. The term probabilistic modeling language is less common, but clearly expresses that inference is not part of the feature set.

FlatPPL is primarily declarative: it describes models, not inference procedures. The scientist writes a model that reads like a simulation recipe: start with a set of parameter values, compute derived quantities, and describe how observations arise from distributions that depend on those parameters. The source model is not an inference procedure or control-flow program. It denotes a static mathematical object that different algorithms can traverse or evaluate in different ways (see below).

FlatPPL does, however, also support likelihood object declarations and density evaluation. Density evaluation defines the semantics of likelihood objects and is also useful for density-based computations within deterministic parts of models. This goes beyond what most probabilistic modeling languages offer, which often have a purely Bayesian focus, but is important for a language that aims to mesh well with formats and frameworks like HS³ and RooFit and to equally support both frequentist and Bayesian settings.

Algorithms can use a probabilistic model in two fundamental ways, commonly called **generative mode** and **scoring mode**:

- **Generative mode** (simulation): traverses the declared model graph forward and draws random values from probability distributions to produce synthetic data.
- **Scoring mode** (density evaluation): given parameters and observed values, calculate log-likelihood or log-posterior density values for frequentist and Bayesian inference methods.

Together, generative and scoring mode form the basis for the full range of statistical workflows: maximum likelihood estimation, profile likelihood ratios, Bayesian posterior sampling, hypothesis testing, model comparison, goodness-of-fit checking, and simulation-based inference.

The key design requirements here are:

1. **Language-independent.** Not tied to a specific programming language. The design must allow for implementation of generative and scoring mode in a wide variety of host languages.
2. **Inference-agnostic.** Must serve both Bayesian and frequentist use cases.
3. **Not tied to a specific engine.** No coupling to particular inference algorithms or computational backends.
4. **Long-lived.** Code lifetimes in HEP have long been measured in decades and data preservation is becoming an increasing concern in many scientific fields. The design must be durable enough to outlast current software and hardware ecosystems.

5. **Expressively sufficient.** Must allow us to express a wide corpus of models across many scientific domains.

Accelerator compatibility. Models that are expressed as a static DAG of bindings — with value shapes that can be inferred at compile time, no loops, no dynamic control flow, no data-dependent shapes, but with explicit support for elementwise operations — map naturally to accelerator-oriented IRs such as MLIR/StableHLO/XLA. Engines targeting high-performance backends (e.g., via JAX in Python or Reactant.jl in Julia) can lower operations on a model, like sampling or density/likelihood evaluation, to these IRs — without fundamental impedance mismatches for the large class of common models with static topology and statically known shapes.

1.4 The case for a new probabilistic language

We surveyed the landscape of probabilistic languages, but no currently available language covers all of our requirements. Some relevant examples are:

Stan (Carpenter et al., 2017) is the strongest candidate for longevity: it has a large and active user and developer community, bindings for multiple languages (R, Python, Julia and others), and solid funding. However:

- Stan is fundamentally Bayesian, and there is no separation between prior and observation model in a Stan model block. This means that there is no access to the likelihood for frequentist settings, and no way to express one as a standalone object.
- The Stan language is tightly coupled to a specific compiler and runtime (stanc → C++); there is no independent second implementation of the language specification, making it difficult to adopt as a language-independent interchange format.
- Stan is a full probabilistic programming language with rich syntax, it cannot function as a serialization format, and there is no export path to one.

SlicStan (Gorinova et al., 2019) introduced compositional, blockless Stan with an information-flow type system for automatic variable classification. The "shredding" approach is relevant to our design. But it remains a Stan dialect, inheriting Stan's Bayesian orientation.

Pyro/NumPyro, Turing.jl, PyMC are embedded in their host languages and tightly coupled to specific inference engines.

GraphPPL.jl (used by RxInfer) separates model specification from inference backend, which is architecturally what we want. But it's Julia-specific and Bayesian-focused.

Hakaru (Narayanan et al., 2016) has elegant semantics built on the Giry monad, expressing programs as measure expressions with support for both frequentist and Bayesian reasoning. However, it does not appear to be actively maintained, and is tied firmly to the Haskell language.

Birch is a standalone PPL transpiling to C++, but more of an academic project without guaranteed longevity.

Two recent research projects from the PL community are tangentially relevant. **LASAPP** (Böck et al., 2024) demonstrates that a cross-PPL abstraction layer is achievable, though its IR is too

minimal for our needs. Fenske et al. (2025) propose a representation-agnostic factor abstraction, but it operates at the inference level, below where a model specification language sits.

1.5 Use cases for FlatPPL

Model representation and evaluation. Models can be directly authored in or converted to FlatPPL and evaluated via FlatPPL implementations/engines. FlatPPL is designed for efficient implementation in multiple host languages and use of accelerator hardware. Once stable, FlatPPL will aim for long-term backward-compatibility.

Model conversion. FlatPPL is designed to be a suitable intermediate stage when converting models between different stochastic languages and formats.

Design and reasoning. FlatPPL is meant to explore a set of mathematical and statistical semantics that is broad, coherent and rigorous, and has the potential to inform design or extension of other probabilistic languages and standards. FlatPPL also comes with a simple and readable canonical syntax that makes it suitable as a reasoning aid in cases where mathematical notation would still be too informal or too dependent on context. The semantics of FlatPPL are independent from this canonical syntax, though.

---

2 Language overview

2.1 FlatPPL in a nutshell

The name **FlatPPL** reflects the language's most distinctive design choices. Probabilistic models are expressed as static graphs of named mathematical objects — variates, measures, functions, and likelihoods — in a single flat namespace with no blocks, no scoping, no function definitions, and no loops or dynamic branching. A FlatPPL document is a sequence of named bindings in static single-assignment (SSA) form. The order of statements is semantically irrelevant; the graph structure is determined by name references, not by textual position. Data is represented by ordinary values (arrays, records, tables).

This simplicity makes FlatPPL amenable to serialization, static analysis, and compilation to accelerator backends, while still being expressive enough to cover a wide range of models across scientific domains. The resulting graph structure is similar to an HS³ JSON document or a RooFit workspace, though FlatPPL concepts like random draws and measure/function reification do not currently exist in HS³ and RooFit. See the profiles and interoperability section on how FlatPPL maps to them.

Creating FlatPPL models. FlatPPL is intended both for direct authoring and as a target representation for models defined in other stochastic languages. Writing FlatPPL documents directly is practical for smaller and medium-sized models and for didactic settings. Users may prefer to use host-language frameworks to author models though, especially larger and complex models, or to use other DSLs to run, exchange and preserve models. FlatPPL is designed to be broad enough to make it a practical target from a wide variety of modeling frontends.

Converting FlatPPL models. FlatPPL defines profiles (see Profiles), specific subsets of the language for easy mapping to target probability languages and formats.

Canonical syntax. FlatPPL comes with a canonical surface form, a simple syntax designed to parse as both valid Python and valid Julia. Due to this, FlatPPL is suited for embedding in Python and in Julia code as a DSL. See the section on the canonical syntax for details.

This document uses the canonical syntax as a notation to define FlatPPL semantics and to present language examples. Note though that the semantics of FlatPPL are independent from this canonical syntax.

Canonical intermediate representation. FlatPPL also comes with a canonical intermediate representation (IR) based on S-expressions, the Flat Probabilistic Intermediate Representation (FlatPIR) (see Intermediate Representation). FlatPIR supports metadata like type and phase annotations and is suitable for automated term-rewriting, enabling code optimization and conversion between FlatPPL profiles. FlatPPL maps directly to and from FlatPIR, enabling round-tripping.

2.2 Implementation targets

The scientific communities where we expect FlatPPL to see most use primarily work in C++/ROOT, Python, and Julia. Each ecosystem brings its own strengths and infrastructure for statistical modeling:

C++ / RooFit. RooFit is the most mature and widely deployed statistical modeling toolkit in high energy physics, but currently lacks some features required in other fields where it could play a role. FlatPPL is likely to target RooFit via HS³ conversion initially, though direct support is also possible. In either case the implementation strategy is evolution, not replacement: non-breaking additions to widen the semantic scope of RooFit and bring it as close to the scope of FlatPPL as feasible. See the HS³/RooFit profile section for details. Stan, as mentioned before, is very powerful but does not cover all of our requirements. Many strictly Bayesian FlatPPL models could be converted to Stan model blocks though and run on the Stan engine. Accelerator support for RooFit seems less likely for now in general.

Python. pyhf covers the HistFactory subset of HS³, zfit has partial support, and pyhs3 provides a first Python HS³ implementation. In regard to FlatPPL there is more room for direct support in the Python ecosystem than in C++/RooFit. JAX offers a natural path to accelerator-oriented execution via MLIR/StableHLO.

Julia. There is only a prototype HS³ implementation in Julia (HS3.jl). Julia has a rich ecosystem of statistics packages like Distributions.jl and MeasureBase.jl that provide an excellent basis for an inference-agnostic implementation of FlatPPL, orthogonal to inference packages like ProfileLikelihood.jl, BAT.jl and others. FlatPPL and HS³ models could be supported in Julia via the same graph engine. The Julia equivalent to JAX is Reactant.jl, like JAX it targets accelerators via MLIR/StableHLO.

2.3 A first example

Before delving into the language more formally, here is a small example to convey the flavor of the FlatPPL language. This high energy physics model describes a simple particle mass measurement where the observed spectrum is a superposition of signal and background events, with a systematic uncertainty on the signal resolution:

```
# Inputs: expected signal and background event counts
n_sig = elementof(reals)
n_bkg = elementof(reals)

# Systematic: uncertain detector resolution
raw_syst = draw(Normal(mu = 0.0, sigma = 1.0))
resolution = 2.5 + 0.3 * raw_syst

# Signal: Gaussian peak at known mass, uncertain resolution
signal_shape = Normal(mu = 125.0, sigma = resolution)

# Background: falling exponential
background_shape = Exponential(rate = 0.05)

# Unbinned data
observed_data = [120.1, 124.8, 125.3, 130.2, 135.7, 142.0]

# Combined intensity: unnormalized superposition (weights = expected event
counts)
intensity = superpose(
    weighted(n_sig, signal_shape),
    weighted(n_bkg, background_shape)
)

# Unbinned model: Poisson process over scalar mass values
events = draw(PoissonProcess(intensity = intensity))

# Observed data and likelihood
L = likelihoodof(lawof(events), observed_data)
```

Reading top to bottom, this is a generative recipe: declare inputs, draw a systematic shift, compute the resolution, define signal and background shapes, combine them as an unnormalized superposition (where the weights encode expected event counts), and draw events from the resulting Poisson process. Since the event space is scalar (mass values), the `PoissonProcess` produces an array variate, and the observed data is a plain array of mass values. (This top-to-bottom reading is for intuition only; semantically the bindings form a dependency graph and may be resolved in any topological order.) The same specification supports generative mode (engines draw synthetic events) and scoring mode (engines compute the log-likelihood at given parameter values).

Note. From a Bayesian perspective, the same model can be read as having a prior `Normal(mu = 0.0, sigma = 1.0)` on `raw_syst`, with `n_sig` and `n_bkg` as externally supplied hyperparameters or model parameters. This illustrates FlatPPL's inference-agnostic design.

2.4 Core concepts

FlatPPL has four kinds of first-class objects.

Abstract values denote real numbers, integers, booleans, complex numbers, fixed-size arrays, and records. They may be deterministic (literal constants, results of ordinary functions, data, external inputs) or stochastic (variates introduced by `draw(...)`). In generative mode, each abstract value evaluates to a single concrete value; for stochastic abstract values, that concrete value is generated randomly once.

Kernels, measures and distributions. Transition kernels are mappings from an input space to measures. FlatPPL does not distinguish between a kernel with an empty interface and a measure: in FlatPPL, such a kernel *is* a measure (see variates and measures). Normalized measures (kernels) are probability measures (Markov kernels), also called probability distributions. Variates can only be drawn from probability measures. Otherwise, FlatPPL treats measures and kernels uniformly in measure algebra (see measure algebra). Variates can be reified as Markov kernels, or probability measures, via `lawof(...)` (see kernels, measures and `lawof`).

Likelihood objects represent the density of a model evaluated at observed data, as a function of the model's input parameters. The observed data is bound to the likelihood object when it is constructed. To prevent a mix-up of likelihood and log-likelihood values, FlatPPL does not treat a likelihood object as a function that returns the one or the other. Instead, (log-)likelihood values are computed via `densityof(L, theta)` and `logdensityof(L, theta)` to make the choice explicit. (See likelihoods and posteriors for the full treatment.)

Functions compute result values from input values in a deterministic fashion. See calling conventions and anonymous functions for details. Values can be reified as functions via `functionof(...)`.

Tuples are ordered bundles of objects. Tuples may contain any objects except for other tuples (see Tuples).

Measures, likelihood objects, functions, and tuples are first-class in the sense that they can be bound to names, passed to operations and referenced by other bindings. However, they may not appear inside arrays, records, or tables.

Modules represent whole FlatPPL documents; each FlatPPL source file is a module. A FlatPPL module can load other modules (via `load_module(module_filename)`) and access objects in loaded modules via dot-syntax scoping (`loaded_module.some_object`). Module objects give access to another namespace, but are not themselves first-class objects in the computational graph: they may not be passed to functions or appear inside data structures. See multi-file models for details.

2.5 Language map

The table below provides a compact overview of the language. Each family name links to the section where the constructs are documented.

Family	Constructs
Special operations	`draw`, `lawof`, `functionof`, `fn`, `fchain`, `bijection`, `elementof`, `valueset`
Interface adaptation	`relabel`
Measure combinators	`weighted`, `logweighted`, `normalize`, `totalmass`, `superpose`, `joint`, `jointchain`, `chain`, `iid`, `truncate`, `pushfwd`
Likelihoods and posteriors	`likelihoodof`, `joint_likelihood`, `densityof`, `logdensityof`, `bayesupdate`
Structural disintegration	`disintegrate`
Higher-order operations	`broadcast`, `broadcasted`, `reduce`, `scan`
Data access and reshaping	`get`, `cat`, `record`, `tuple`, `preset`, `fixed`, `all`, `filter`, `selectbins`, `reverse`
Array and table generation	`table`, `rowstack`, `colstack`, `partition`, `linspace`, `extlinspace`, `fill`, `zeros`, `ones`, `eye`, `onehot`, `load_data`
Binning and interpolation	`bincounts`, `interp_pwlin`, `interp_pwexp`, `interp_poly2_lin`, `interp_poly6_lin`, `interp_poly6_exp`
Shape functions	`polynomial`, `bernstein`, `stepwise`
Math and logic	`identity`, `exp`, `log`, `pow`, `sqrt`, `abs`, `sin`, `cos`, `min`, `max`, `div`, `mod`, `ifelse`, `land`, `lor`, `lnot`, `lxor`
Operator functions	`add`, `sub`, `mul`, `divide`, `neg`, `equal`, `unequal`, `lt`, `le`, `gt`, `ge`
Complex arithmetic	`complex`, `real`, `imag`, `conj`, `abs2`, `cis`
Reductions	`sum`, `product`, `length`
Norms and normalization	`l1norm`, `l2norm`, `l1unit`, `l2unit`, `logsumexp`, `softmax`, `logsoftmax`

Family	Constructs
Distributions	`Normal`, `Poisson`, `PoissonProcess`, `BinnedPoissonProcess`, `Exponential`, `Dirichlet`, …
Fundamental measures	`Lebesgue`, `Counting`, `Dirac`
Module operations	`load_module`, `flatppl_compat`
Constants	`true`, `false`, `inf`, `pi`, `im`
Predefined sets	`reals`, `posreals`, `nonnegreals`, `unitinterval`, `posintegers`, `nonnegintegers`, `integers`, `booleans`, `complexes`, `anything`
Set constructors	`interval`, `cartprod`, `cartpow`, `stdsimplex`
Selectors and operators	`all` (slicing), `in` (membership)

2.6 A tour of FlatPPL

The following code blocks illustrate the main language features. Each construct is explained in detail in later sections. These snippets are independent and do not form a single model.

2.6.1 Values and collections

Scalars, arrays, nested arrays, matrices, records, and basic operations:

```
# Scalars
x = 3.14
n = 42
b = true

# Collections
v = [1.0, 2.0, 3.0]
nested = [[1, 2], [3, 4]]
M = rowstack([[1, 2, 3], [4, 5, 6]])
r = record(mu=3.0, sigma=1.0)

# Indexing, field access, slicing
y = A[i]
z = A[i, j]
w = r.mu
col_j = M[:, j]

# Decomposition into named scalars
a, b, c = draw(MvNormal(mu = mean, cov = cov_matrix))
p, q = some_record
```

```
l, m, n = some_tuple

# Arithmetic, comparisons, function calls
rate = efficiency * mu_sig + background
is_positive = x > 0
y = exp(x)
z = ifelse(is_positive, a, b)
combined = cat(record1, record2)
joined = cat(array1, array2)
```

2.6.2 Function calls

Calling conventions:

```
Normal(mu = 0, sigma = 1)           # keyword
Normal(record(mu = 0, sigma = 1))   # record auto-splatting
exp(x)                              # positional
```

2.6.3 Complex arithmetic

Constructing and calculating with complex values:

```
# Complex construction
z1 = complex(3.0, 2.0)
z2 = 3.0 + 2.0 * im               # equivalent
phase = cis(3 * pi / 4)           # unit-modulus complex from angle

# Complex arithmetic
A_total = A_sig * coupling + A_bkg

# Squared modulus (real-valued result)
intensity = abs2(A_total)

# Decomposition and conjugation
x = real(z1)
y = imag(z1)
z_bar = conj(z1)
```

2.6.4 Draws, measures, and the stochastic core

`draw`, `lawof`, and `functionof` bridge between values, measures, and functions:

```
# Inputs
mu = elementof(reals)
sigma = elementof(interval(0.0, inf))

# Random draw from a distribution
a = draw(Normal(mu = mu, sigma = sigma))
```

```
# Extract the distribution governing a value
M = lawof(a)

b = 2 * a + 1

# Reify a deterministic sub-DAG as a function
f = functionof(b)

# With explicit input naming and ordering
f_named = functionof(b, x = a)

# Stochastic sub-DAG as a kernel
K = lawof(c)
```

2.6.5 Broadcasts

`broadcast` applies functions or kernels elementwise over arrays and tables:

```
# Function over array (keyword binding)
C = broadcast(f_named, x = A)

# Same, positional
C = broadcast(f_named, A)

# Kernel over array
D = draw(broadcast(K, a = A))
```

2.6.6 Value-level operations

Accessing and renaming values, and transforming measures based on that:

```
# Element and subset access
field_a = get(some_record, "a")
sub = get(some_record, ["a", "c"])

# Array to record conversion
named = relabel(some_array, ["a", "b", "c"])

# Structural relabeling of a measure
mvmodel = relabel(MvNormal(mu = some_mean, cov = some_cov), ["a", "b", "c"])

# Variable transformation
log_normal = pushfwd(functionof(exp(x), x = x),
    Normal(mu = 0, sigma = 1))

# Projection (marginalizes out b)
marginal_ac = pushfwd(fn(get(_, ["a", "c"])), mvmodel)
```

2.6.7 Measure algebra and composition

Combining, reweighting, and transforming measures some more:

```
# IID draws
xs = draw(iid(Normal(mu = 0, sigma = 1), 100))

# Additive rate superposition
sp = superpose(weighted(n_sig, sig), bkg)

# Normalized mixture
mix = normalize(superpose(
    weighted(0.7, M1), weighted(0.3, M2)))

# Independent joint
j = joint(M1, M2)

# Marginalizing composition
pp = chain(prior, forward_kernel)

# Hierarchical joint (retains both variates)
hj = jointchain(
    pushfwd(fn(relabel(_, ["a"])), M1),
    pushfwd(fn(relabel(_, ["b"])), K_b))

# Truncated (unnormalized) measure
positive_normal = truncate(Normal(mu = 0, sigma = 1),
    interval(0, inf))

# Fundamental measures and density-defined distributions
leb = Lebesgue(support = reals)
bern = fn(bernstein(coefficients = [c0, c1, c2], x = _))
smooth_shape = normalize(weighted(bern, Lebesgue(support = interval(lo, hi))))
```

2.6.8 Anonymous functions

The `fn(...)` form wraps a hole expression — an expression containing `_` — to create an anonymous function with positional parameters:

```
# Single hole — one-argument function
poly = fn(polynomial(coefficients = [a0, a1, a2], x = _))
squared = fn(pow(_, 2))

# Multi-hole: two-argument anonymous function
ratio_sq = fn(pow(_ / _, 2))
```

2.6.9 Interpolation, binning, and systematic variations

Constructors for binned models and HistFactory-style yield arithmetic:

```
edges = linspace(0.0, 10.0, 5)
counts = bincounts(edges, event_data)

# Binned observation model via pushforward
binned_model = pushfwd(fn(bincounts(edges, _)),
    PoissonProcess(intensity = M_intensity))

# Interpolation for systematic variations
kappa = interp_poly6_exp(0.95, 1.0, 1.05, alpha)
morphed = interp_poly6_lin(tmpl_dn, nominal, tmpl_up, alpha)
```

2.6.10 Data

Data is represented by ordinary values — no special data type:

```
observed_counts = [5, 12, 8, 3]
data_table = table(a = [1.1, 1.2], b = [2.1, 2.2])
```

2.6.11 Presets

Advisory parameter/input values for use with a compatible function, kernel, or likelihood:

```
# Parameter starting values (advisory, not part of model semantics)
starting_values = preset(mu_sig = 1.0, raw_syst = fixed(0.0), n_bkg = 50.0)
```

2.6.12 Analysis: likelihoods and posteriors

Likelihood construction, combination, and posterior construction:

```
L = likelihoodof(lawof(obs), data)
R = interval(2.0, 8.0)
L_sub = likelihoodof(normalize(truncate(lawof(obs), R)), filter(fn(_ in R),
data))
L_total = joint_likelihood(L1, L2)

# Unnormalized posterior
posterior = bayesupdate(L, prior)

# Deterministic function composition
pipeline = fchain(calc_kinematics, apply_cuts)
```

2.6.13 Modules and interface adaptation

Module loading and parameter renaming:

```
# Load a module and optionally bind some of its inputs
sig = load_module("signal_channel.flatppl", mu = signal_strength, theta =
nuisance)
```

```
sig_model = sig.model
L_sig = likelihoodof(sig.model, sig.data)
```

---

3 Value types and data model

FlatPPL has a small, fixed set of value types. This section defines what kinds of values exist in the language, their invariants, and how they interact. Operations on values are documented in built-in functions.

3.1 Scalar types

Real. Floating-point numbers like `3.14`, `-0.5`, `1e-3`.

Integer. Integer numbers like `42`, `0`, `-7`.

Bool. `true` or `false` (lowercase). In arithmetic contexts, `false` is promoted to zero and `true` to one, permitting expressions such as `true + true`, `3 * false`, and `sum(mask)` to count true entries. Conditional and logical constructs (`ifelse`, `land`, `lor`, `lnot`, `lxor`) strictly require boolean arguments; zero and one are not implicitly converted to booleans.

Complex. A complex number. Constructed via `complex(re, im)` or via arithmetic with the imaginary unit `im`:

```
z1 = complex(3.0, 2.0)
z2 = 3.0 + 2.0 * im           # equivalent
phase = cis(3 * pi / 4)       # unit-modulus complex from angle
```

When a real and a complex value meet in arithmetic, the real is promoted to complex with zero imaginary part.

Scalar value categories and sets. FlatPPL distinguishes boolean, integer, real, and complex scalar values operationally. In particular, conditionals and logical operators require boolean values. However, the predefined value sets satisfy the canonical inclusions `booleans` $\subset$ `integers` $\subset$ `reals`, and there is a canonical embedding of `reals` into `complexes`. Arithmetic may use these canonical embeddings implicitly where specified by the language.

3.2 Predefined constants

Name	Type	Description
`true`, `false`	Bool	Boolean constants
`inf`	Real	Positive infinity ($+\infty$). Used in `interval`, `extlinspace`, `truncate`
`pi`	Real	The mathematical constant $\pi \approx 3.14159\ldots$

Name	Type	Description
`im`	Complex	The imaginary unit i ($i^2 = -1$). Equivalent to `complex(0.0, 1.0)`
`reals`	Set	The real numbers, with $\pm\infty$ admitted (see note below). Default support for `Lebesgue`
`posreals`	Set	The positive reals including $+\infty$: $(0, +\infty]$
`nonnegreals`	Set	The non-negative reals including $+\infty$: $[0, +\infty]$
`unitinterval`	Set	The unit interval $[0, 1]$
`posintegers`	Set	The positive integers $\{1, 2, 3, \ldots\}$
`nonnegintegers`	Set	The non-negative integers $\{0, 1, 2, \ldots\}$
`integers`	Set	The set of all integers ($\mathbb{Z}$). Default support for `Counting`
`booleans`	Set	The set $\{\text{false}, \text{true}\}$
`complexes`	Set	The set of all complex numbers ($\mathbb{C}$)
`anything`	Set	Generic placeholder set for untyped interfaces (see sets)

Note on infinities. `posreals`, `nonnegreals`, and `reals` admit `inf` (and, for `reals`, `-inf`) as legal values. Strictly speaking, these are subsets of the extended reals $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$, not of $\mathbb{R}$. This is a deliberate choice for compatibility with common numerical and statistical libraries. When FlatPPL refers to `Lebesgue(support = reals)`, the reference measure is the ordinary Lebesgue measure on the finite-real part; the points $\pm\infty$ carry zero Lebesgue mass. Arithmetic on infinities follows IEEE 754 conventions.

The selector `all` and the hole token `_` are syntactic elements, not value constants; they are documented in calling conventions and anonymous functions.

3.3 Arrays

Arrays are fixed-size, ordered, n-dimensional collections of scalar values (real, integer, boolean and complex values) or arrays.

Literal one-dimensional arrays are denoted as `[1.0, 2.0, 3.0]` and may contain arbitrary valid FlatPPL expressions that evaluate to allowed element types (e.g. `[a, b, 2 * c]`).

One-dimensional arrays of scalars act as vectors for linear algebra (see built-in functions). Vectors of vectors are not interpreted as matrices implicitly, but can be turned into matrices explicitly using `rowstack` or `colstack` (see array operations).

FlatPPL supports standard linear algebra operations (addition, multiplication) on scalars, vectors, and matrices.

3.4 Records

Records comprise ordered named fields, written as `record(name1=val1, name2=val2, ...)`. Field values may be scalars or arrays, but not records. Field access uses dot syntax: `r.name1` (lowers to `get(r, "name1")`). Field order is part of the record's identity: `record(a=1, b=2)` and `record(b=2, a=1)` are distinct values. This is significant for alignment with parameter spaces and for deterministic serialization. Fields are accessed by name, not by position — `get(r, i)` is not supported to avoid ambiguity with row indexing on tables.

3.5 Presets

Presets are records tagged as suitable parameter or input values. Presets are advisory and not tied to a particular function, kernel, or likelihood. It is up to users and tooling to pair presets with compatible interfaces and decide how to use them, for example as reference points, starting values for optimizers, or similar.

Presets are written `preset(name1=val1, name2=val2, ...)`. Presets accept the same field value types as records. However, values may be wrapped in a `fixed(...)` marker to indicate that they are intended to be held constant, e.g. during optimization. `fixed` may only appear at the top level of a `preset(...)`. Presets may not be nested.

For example

```
starting_values = preset(a = 2.0, b = [4, 5, 6], c = fixed(8.0))
```

informs users and tools that `starting_values` may be a good choice of test point or starting point for functions, kernels or likelihoods that take parameters `a`, `b`, and `c`, and that if this preset is chosen, `c` should be held constant while `a` and `b` are varied.

Within FlatPPL, a `preset` object is semantically equivalent to a record, and converts to a record in any context that expects a record as an input. The `preset` annotation and `fixed` markers are lost at that point, they do not propagate.

3.6 Tables

Tables are datasets that consist of named columns of equal length. Table columns must be vectors, as using higher-dimensional arrays as columns would require a leading-axis convention for row iteration and broadcasting, which FlatPPL intentionally avoids.

Tables are constructed from columns via `table(col1 = [...], col2 = [...])`:

```
events = table(mass = [1.1, 1.2, 1.3], pt = [45.2, 32.1, 67.8])
```

Implementations may choose whichever table realization and memory layout they prefer, also on a case-by-case basis.

Tables can also be constructed from records of equal-length vectors via `table(r)` and converted to such records via `record(t)`, due to FlatPPL auto-splatting semantics.

Indexing. Tables support both column and row access:

- Column access by field name: `t.colname`, equivalent to `get(t, "colname")`, returns the column with that name as a vector.
- Row access by integer index: `t[i]`, equivalent to `get(t, i)`, returns the `i`-th row as a record (`i` starts at 1).

`length(t)` returns the number of table rows.

Broadcasting. When a table is passed to `broadcast`, it is traversed row-wise and each row treated as a record passed to the function used in the broadcast.

Data carriers by model shape. FlatPPL uses ordinary values as data carriers:

- **Single scalar datum** → scalar value
- **Single structured datum** → record or array
- **Unbinned scalar event sample** → plain array
- **Unbinned multivariate event sample** → table
- **Binned count data** → plain count array

3.7 Sets

FlatPPL has a limited notion of sets, used to specify input domains, supports, truncation regions, and analysis regions. The predefined sets are:

- `reals` — $\mathbb{R}$, the set of all real numbers.
- `posreals` — $(0, +\infty]$, the positive reals including $+\infty$.
- `nonnegreals` — $[0, +\infty]$, the non-negative reals including $+\infty$.
- `unitinterval` — $[0, 1]$, the unit interval.
- `posintegers` — $\{1, 2, 3, \ldots\}$, the positive integers.
- `nonnegintegers` — $\{0, 1, 2, \ldots\}$, the non-negative integers.
- `integers` — $\mathbb{Z}$, the set of all integers.
- `booleans` — $\{\text{false}, \text{true}\}$.
- `complexes` — $\mathbb{C}$, the set of all complex numbers.
- `anything` — a broad placeholder set for generic interfaces (e.g., anonymous functions via holes). Not formally the union of all other sets; it signals that no specific type constraint is imposed.

Additional sets may be constructed using the following language constructs:

Interval. `interval(lo, hi)` denotes the closed interval $[lo, hi]$.

Cartesian product. `cartprod(S1, S2, ...)` produces a Cartesian product of sets `S1`, `S2`, etc., mirroring `joint(M1, M2, ...)` for measures. The result is the set of arrays whose elements lie in the respective component sets. For example, `cartprod(reals, posreals)` is the set of 2-element arrays with the first element in $\mathbb{R}$ and the second in $(0, \infty)$.

The keyword form `cartprod(a = S1, b = S2, ...)` produces a set of records with field `a` in `S1`, field `b` in `S2`, etc., mirroring `joint(a = M1, b = M2, ...)`.

Cartesian power. `cartpow(S, m, n, ...)` produces the Cartesian power $S^{m \times n \times \cdots}$, mirroring `iid(M, m, n, ...)` for measures. So `cartpow(reals, 3)` represents $\mathbb{R}^3$.

Standard simplex. `stdsimplex(n)` denotes the standard $(n-1)$-dimensional probability simplex $\Delta_{n-1} = \left\{x \in \mathbb{R}^n : x_i \geq 0, \sum_i x_i = 1\right\}$. `Lebesgue(support = stdsimplex(n))` is the $(n-1)$-dimensional Hausdorff measure on the simplex, embedded in $\mathbb{R}^n$: it measures surface area within the simplex and assigns zero mass to sets that do not intersect it.

`relabel` applies to set products in the same way as to measures (see interface adaptation).

Sets that govern values. `valueset(x)` returns the canonical value set associated with node `x`:

- For `x = elementof(S)`, `valueset(x)` is `S`.
- For `x = draw(M)`, `valueset(x)` is the measurable set of `M`.
- For deterministically computed nodes, `valueset(x)` returns a conservative superset of the values that `x` can take (since the exact set is often not tractable).

Note: `valueset` is a low-level language construct used when lowering `functionof` or `lawof` with boundary inputs. User-level code should typically use `elementof(...)` and specify sets explicitly.

3.8 Beyond values

Measures, likelihood objects, functions, tuples, and modules are also first-class objects in FlatPPL — they can be bound to names, passed to combinators, and referenced by other bindings. However, they are not value types: they may not appear inside arrays, records, or tables. See core concepts for details.

---

4 Language design

This section details the semantics of FlatPPL's core constructs: modules, objects, and callables, and built upon them variates, measures, deterministic and stochastic graphs, and more.

4.1 Objects, expressions, names and modules

The FlatPPL language consists of **objects** (**measures** or **values** like numbers, arrays, records and tables). Objects include **callables** (value functions, constructors, transition kernels and special operations) that operate on objects.

Ordinary **callables** have named inputs and a single output; that output may be a tuple bundling multiple components (see Tuples). Special operations are callables that handle inputs in a different way and typically provide higher-level semantics. The output of any callable depends deterministically on its inputs and calls may not have any side effects. No callables may have nullary inputs, as this would make them equivalent to known values.

Numerical precision (e.g., 32-bit vs. 64-bit floating point) is not specified by FlatPPL; the choice is left to implementations and their users.

A FlatPPL **module** is an unordered set of bindings of names to expressions. Expressions are single or nested calls that bind expressions (literal or by name reference) to inputs of callables.

FlatPPL is loop-free and has no block structure, so a module is implicitly a directed acyclic graph (**DAG**), which may not be fully connected. The nodes in that graph are the named and unnamed (results of) calls, the edges are the connections between outputs and inputs of calls.

The graphs of several modules can be combined, see Module composition below for details.

Note: Record field names and table column names are local to their object and not part of the global module namespace, nor are the argument names of functions and kernels.

4.2 Binding names

Public bindings. Names that do not begin with an underscore are public: they form the interface of a FlatPPL module. They must match the regular expression `^[A-Za-z][A-Za-z0-9_]*$`.

Private bindings. Binding names that begin with a single underscore and do not end with an underscore (regular expression `^_[A-Za-z]([A-Za-z0-9_]*[A-Za-z0-9])?$`), e.g. `_tmp`, are private to a module. They are not part of the module's public interface and may be eliminated, inlined, renamed, or otherwise not preserved by tooling such as term-rewriting or dead-code elimination.

Auto-generated names. Names starting with a double underscore (regular expression `^__[A-Za-z0-9][A-Za-z0-9_]*$`) are reserved for automatically generated module-level binding names. Like other underscore-prefixed names they are private and elidable. Auto-generated binding names with a purely numerical ID are denoted in hexadecimal form with a `__0x` prefix in the canonical FlatPPL and FlatPIR syntax (regular expression `^__0x[0-9a-f]+$`), but may be encoded differently in other representations of the language.

Placeholder names. Names starting and ending with a single underscore (regular expression `^_[A-Za-z]([A-Za-z0-9_]*[A-Za-z0-9])?_$`) are reserved for placeholder variables inside `functionof` and `lawof` (see placeholders and holes). They must not be used for module-level bindings.

4.3 Calling conventions

Nullary calls (`f()`) are not allowed.

All ordinary callables — built-in or user defined value functions, constructors or transition kernels — accept arguments in two or three forms (denoted here in the canonical syntax):

- **Keyword arguments** (named arguments): `f(a = x, b = y, ...)`. Arguments are bound to inputs by name, the order of the arguments is not relevant.
- **Auto-splatting** (of records and tables columns): `f(record(a = x, b = y, ...))` and `f(table(a = x, b = y, ...))` are equivalent to `f(a = x, b = y, ...)`. The order of fields or columns is not relevant. A call with field or column names that do not match the callable's argument names is a static error. Auto-splatting is shallow.

- **Positional arguments**: `f(x, y, ...)`. Positional arguments are accepted only if the callable has ordered inputs, so that the arguments can mapped to the inputs in order.

All built-in ordinary callables have a defined input order accept both positional and keyword arguments.

Special operations have zero to three distinguished, unnamed, ordered inputs of fixed arity. They and may have additional variadic named or unnamed inputs, the order of which may or may not be significant. The total number of inputs is never zero:

- `elementof`, `external`, `draw`: One distinguished input.
- `vector`: Unnamed variadic inputs with significant order.
- `tuple`: Unnamed variadic inputs with significant order (minimum two).
- `record` and `table`: Named variadic inputs with significant order.
- `functionof` and `lawof`: One distinguished input, plus optional variadic named inputs with significant order.
- `broadcast`: One distinguished input for the function to be broadcast, plus named or unnamed inputs that match the inputs of that function.
- `broadcasted`: One distinguished input.
- `cat`, `zeros`, `ones`, `fchain`, `chain`: Variadic unnamed inputs with significant order.
- `cartprod`, `joint`, `jointchain`: Variadic unnamed or named inputs with significant order.
- `get`, `fill`, `cartpow`, `iid`: One distinguished input plus variadic unnamed input with significant order.
- `superpose`: Variadic unnamed inputs with no significant order.
- `load_module`: One distinguished input plus optional variadic named inputs with no significant order.
- `load_data`: One distinguished input plus optional variadic named inputs with significant order.
- `checked`: Two distinguished inputs.

4.4 Tuples

Some operations produce a single output that naturally groups several distinct components — e.g. a kernel and its base measure together, or an updated RNG state alongside a generated value. **Tuples** package such outputs as an ordered, fixed-length bundle of FlatPPL objects.

The surface form `(a, b, c)` lowers to `tuple(a, b, c)`. Tuples must contain at least two components, so `()`, `(x,)` are not allowed. Tuple elements are accessed via `t[i]`, lowering to `get(t, i)`, with a non-negative integer literal index. Decomposition as in `a, b, c = (...)` is positional.

Tuples are objects, not values. They have no `valueset`, are not drawn from measures, and are not part of the measure algebra. Specifically:

- A tuple may not appear inside another tuple, array, record, table, or preset.
- `elementof(...)` and `external(...)` may not produce tuples.
- Measures, kernels, and likelihoods never use tuples as their domain.
- `==`/`equal` does not compare tuples.
- Tuples do not auto-splat like records and tables do.

Tuples otherwise flow like other objects. They may be bound to names, passed to callables that accept them, returned from user-defined functions, and decomposed or projected.

4.5 Variates and measures

FlatPPL distinguishes **variates** from **measures** and **kernels**. A variate represents a specific value — one realization in any given evaluation of the model. A measure or kernel, by contrast, represents the entire distribution over possible values. More formally, measures are monadic while variates are not.

Keeping variates and measures distinct matters because arithmetic means different things for each: In mathematics, $2 \cdot x$ transforms a variate (producing a new variate with twice the value), while $2 \cdot \mu$ rescales a measure's total mass. FlatPPL supports both via different syntax — arithmetic on variates, `weighted(...)` on measures.

A binding of the form `c = f(a, b)` introduces a **deterministic node** in the computational DAG. A binding of the form `x = draw(Normal(mu = c, sigma = s))` introduces a **stochastic node**. In generative mode, a stochastic node yields a sampled value; in scoring mode, it contributes a density term that is either evaluated (if observed) or marginalized out (if latent).

FlatPPL intentionally supports two equivalent mechanisms to express stochastic computations:

1. **Stochastic-node notation** expresses models as a mix of deterministic computations and `draw` statements, reading like a generative recipe.
2. **Measure-composition notation** writes models as a mix of deterministic computations and measure algebra, using `weighted`, `joint`, `jointchain`, `chain`, `pushfwd`, and related operations to combine and transform measures.

Both can be used together in a FlatPPL module, but they map to different types of probabilistic coding systems. Stochastic-node notation mirrors probabilistic programming languages like Stan and Pyro, while measure composition mirrors the HS³ and RooFit approach. By supporting both approaches, FlatPPL can be emitted from both types of systems. Term-rewriting via FlatPIR can raise and lower code to match either of them. This enables FlatPPL and FlatPIR to act as an interoperability platform.

4.6 Internal parameters and external inputs

FlatPPL declares unresolved values via two special operations:

- `elementof(S)` — declares a module-internal parameter that takes a specific value during a single evaluation of a subgraph that contains it; the value may vary between evaluations (e.g., during parameter inference).
- `external(S)` — declares a module-external input. The value must be supplied by applications that use the module or by modules that load the containing module and bind this input to a fixed value in their own namespace. Module inputs can be thought of as hyperparameters and their values don't change between subgraph evaluations.

```
n_dims = external(posintegers)
mu = elementof(reals)
sigma = elementof(interval(0.0, inf))
dist = iid(Normal(mu = mu, sigma = sigma), n_dims)
x = draw(dist)
y = 2 * x
```

The distinction between `external` and `elementof` determines phase classification (see below), closure behavior in `functionof`/`lawof` (see application and reification), and cross-module binding rules for `load_module` (see Multi-file models).

4.7 Phases

FlatPPL classifies every binding into one of three phases by ancestor analysis:

- **fixed** — no `elementof(...)` ancestor and no `draw(...)` ancestor, but may have `load_data(...)` ancestors.
- **parameterized** — at least one `elementof(...)` ancestor, no `draw(...)` ancestor.
- **stochastic** — at least one `draw(...)` ancestor.

External inputs and loaded data (length and content) are fixed; `elementof` inputs are parameterized; `draw` nodes are stochastic. Phase propagates through the DAG: a binding's phase is the dominant of its ancestors' phases (stochastic > parameterized > fixed).

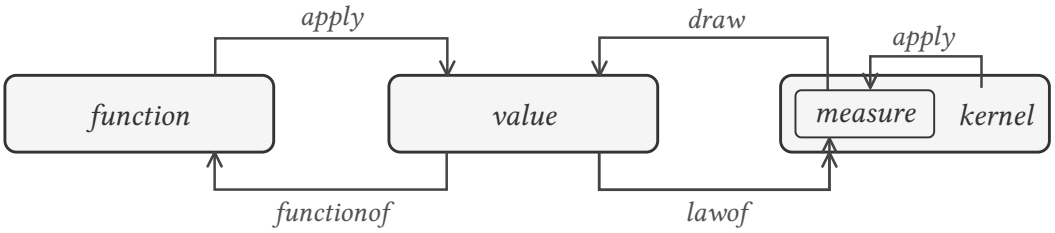
Records, arrays, tables, and tuples may bundle components of differing phases; under the ancestor rule the container carries the joined phase, and projections inherit it. This is conservative — a projection such as `r.a` technically depends on the entire record's ancestors, even though its value is just one field. Engines may sharpen this by flattening projections with statically known selectors (`r.field`, `t[i]` with integer literal index, or the corresponding `get(...)` and decomposition forms) before phase or closure analysis, which recovers the selected component's phase directly.

Both fixed and parameterized bindings are deterministic, but their values have different life cycles: A FlatPPL module can be thought of as having an initialized state, where external inputs have been set and data and referenced modules have been loaded. Fixed values are the values that are given or deterministically computable at this point and so do not change after module initialization. Parameterized values differ between evaluations of the same subgraph (e.g. of a likelihood) of the initialized module, given different inputs. Note that this is a mental model, applications are not required to use an explicit initialization state to implement these semantics.

Phase governs closure behavior (see application and reification) and load-time binding rules (see Multi-file models).

4.8 Application and reification

FlatPPL provides operations that turn subgraphs into first-class objects and vice versa.



A function represents a reified deterministic DAG, either implicit (built-in) or explicitly constructed. Ordinary function application `y = f(a, b, ...)` introduces a deterministic node `y` into the graph. `functionof(y)` goes in the opposite direction: it reifies the ancestor subgraph of a given value `y` as a first-class function.

Conversely, a probability measure represents a reified stochastic DAG, either an implicit (built-in) or explicit one. `x = draw(m)` introduces a stochastic node `x` by drawing a variate from a normalized measure (i.e. a probability measure) `m`. In the other direction, `m = lawof(x)` reifies the ancestor subgraph of a given value `x` as a probability measure or Markov kernel, depending on whether there are free inputs.

By default, `lawof` and `functionof` trace the ancestor subgraph of a given node back to `elementof` nodes; these become inputs of the reified callable. Fixed ancestors (including `external(...)` and `load_data(...)` nodes) are closed over and do not become inputs. `functionof` does not allow for intermediate stochastic ancestors (its full sub-DAG must be deterministic) while `lawof` permits them. Both can be called with additional keyword arguments to designate and label boundary nodes, stopping the trace there so that these nodes become the inputs of the kernel or function under their new names.

4.8.1 Deterministic functions and `functionof`

Consider a simple deterministic computation:

```
c = a + b
d = a - b
e = c * d
```

Here `e` is a specific value during any given evaluation of the code. But the computation that produces `e` from `a` and `b` is useful in its own right: we might want to apply it elementwise over arrays, or use it as a transformation in `pushfwd`. The name `e` refers to a value, not to the computation that produced it, so we need a way to extract the computation as a first-class function:

```
f = functionof(e)                    # f: {a, b: Real} → Real
C = broadcast(f, a = A, b = B)       # apply f elementwise over arrays A, B
```

`functionof(e)` captures the entire computation leading to `e` — the sub-DAG that contains `e` and all its ancestors — as a reusable function object. The sub-DAG must be fully deterministic and so must not contain any `draw` nodes.

The argument names of the resulting function are the names of the leaf nodes of the reified sub-DAG; the input nodes of the function are decoupled from these leaf nodes. Fixed ancestor nodes are closed over and not exposed as inputs. As the graph nodes are not ordered, the function only supports keyword arguments, not positional arguments.

The output type of the reified function matches the type of the argument of `functionof`:

```
f = functionof(e)                                    # scalar output
f = functionof(record(x = something, y = other))     # record output
f = functionof([something, other])                    # array output
f = functionof((something, other))                   # tuple output
```

Boundary inputs may also be tuples (`functionof(..., t = some_tuple_expr)`), in which case the reified function takes a tuple argument.

Specifying reification boundaries. Sometimes only a selected part of the ancestor sub-DAG should be reified. In our example, `e` depends on `c` and `d`, which in turn depend on `a` and `b`. If we want the function represented by the subgraph that starts at `a` and `d` — ignoring how `d` was computed from `a` and `b` — we can specify *boundary inputs* that stop the ancestor backtrace early:

```
g = functionof(e, p = a, q = d)     # g: {p, q: Real} → Real
M2 = pushfwd(g, some_measure)       # transform a measure over (p, q)
```

The keyword arguments `p = a, q = d` declare that the trace stops at nodes `a` and `d`, which become the inputs of `g` under the new names `p` and `q`. The computation from `a` and `b` to `d` is excluded — `g` only contains the path from `a` and `d` to `e`.

If boundary inputs are specified, the reified function supports positional arguments in addition to keyword arguments; their order is determined by the order in which the boundary inputs are specified. Either all inputs must be specified explicitly or none. A specified boundary node `a` can be thought of as being substituted with a new node, generated via `elementof(valueset(a))`, in the reified graph.

The function argument names do not have to differ from the boundary node names:

```
h = functionof(e, a = a, d = d)     # h: {a, d: Real} → Real
```

The resulting function `h` now has arguments named `a` and `d`, but these are local to the function and decoupled from the original nodes `a` and `d`.

Identity law. `functionof(f(a, b), a = a, b = b)` is equivalent to `f`.

4.8.2 Kernels, measures and `lawof`

`lawof(x)` is the stochastic counterpart of `functionof`: it reifies the ancestor sub-DAG of `x` as a **measure** or **kernel**. The same boundary input mechanism and all-or-nothing rule for specifying boundary nodes apply. The key difference is that the sub-DAG may be stochastic and so may (but does not have to) contain `draw` nodes.

Consider this Bayesian example:

```
theta1 = draw(Normal(mu = 0.0, sigma = 1.0))
theta2 = draw(Exponential(rate = 1.0))
a = 5.0 * theta1
b = abs(theta1) * theta2
obs = draw(iid(Normal(mu = a, sigma = b), 10))

joint_model = lawof(record(theta1 = theta1, theta2 = theta2, obs = obs))
prior_predictive = lawof(record(obs = obs))
prior = lawof(record(theta1 = theta1, theta2 = theta2))
forward_kernel = lawof(record(obs = obs), theta1 = theta1, theta2 = theta2)
```

Here we create the following probability measures and kernels:

- `joint_model` is the joint probability distribution over parameters and observation. `joint_model` is equivalent to `jointchain(prior, forward_kernel)`.
- `prior_predictive` is the probability distribution of the observation obtained by marginalizing over `theta1` and `theta2` — they are internal stochastic nodes in the traced sub-DAG, not boundary inputs, so `lawof` integrates them out. `prior_predictive` is equivalent to `chain(prior, forward_kernel)`.
- `prior` is the probability distribution of the parameters `theta1` and `theta2`.
- `forward_kernel` is the Markov kernel of the forward model; it maps values for `theta1` and `theta2` to probability distributions of the observation.

Relationship to `functionof`. On a purely deterministic sub-DAG, `lawof` returns a Dirac kernel while `functionof` returns the deterministic function itself.

Engines are not required to compute marginals eagerly; they may resolve them lazily or symbolically when the measure is consumed.

Identity law. `lawof(draw(m))` is equivalent to `m`, and `lawof(draw(k(a, b)), a = a, b = b)` is equivalent to `k`.

4.9 Interface adaptation

FlatPPL provides `relabel` for structural renaming of outputs. At the value level, `relabel` assigns or renames fields on scalars, arrays, records, and tables:

```
v = relabel([1.0, 2.0, 3.0], ["x", "y", "z"])
# equivalent to:
v = record(x = 1.0, y = 2.0, z = 3.0)
```

and renames record fields and table columns:

```
v = relabel(record(a = 1.0, b = 2.0, c = 3.0), ["x", "y", "z"])
# equivalent to:
v = record(x = 1.0, y = 2.0, z = 3.0)
```

and wraps a scalar into a single-field record:

```
v = relabel(1.0, ["x"])
# equivalent to:
v = record(x = 1.0)
```

The same output-side renaming lifts directly to sets, functions, measures, and kernels:

```
named_S = relabel(cartpow(reals, 3), ["x", "y", "z"])
named_f = relabel(f, ["x", "y", "z"])
named_M = relabel(M, ["x", "y", "z"])
named_K = relabel(K, ["x", "y", "z"])
```

For functions, `relabel(f, names)` is post-composition with `relabel` on the function result; for measures it is equivalent to `pushfwd(fn(relabel(_, names)), M)`; for kernels it acts on the output measures.

See built-in functions for full reference documentation on `relabel`.

4.10 Function composition and annotation

`fchain(f1, f2, f3, ...)` composes deterministic functions left-associatively: `fchain(f1, f2, f3)(x)` equals `f3(f2(f1(x)))`.

`fchain` combines well with auto-splatting: if `f1` returns a record and `f2` accepts keyword arguments matching the record fields, the two functions compose directly. `fchain` is the deterministic analogue of `chain`.

`bijection(f, f_inv, logvolume)` annotates a function `f` with its inverse `f_inv` and the log-volume-element `logvolume` of the forward map. The result is semantically identical to `f`, but engines can use the inverse and volume element when computing densities of pushforward measures. `logvolume` may be a function or a scalar (`0` for volume-preserving maps). See pushfwd for examples.

4.11 Placeholders and holes

4.11.1 Placeholder variables

Creating functions and kernels with boundary inputs via `functionof` and `lawof` requires the creation of unique global variable names. Placeholder variables are special variable names of the form `_name_` (leading and trailing underscore) that are local to a `functionof(...)` and `lawof(...)` and can be thought of as creating unique global input of `elementof(anything)` implicitly. All placeholders must appear both in the expression to be reified and the boundary input keyword arguments.

For example

```
f = functionof(_a_ * b + _c_, a = _a_, c = _c_)
```

is equivalent to:

```
_tmp1 = elementof(anything)
_tmp2 = elementof(anything)
f = functionof(_tmp1 * b + _tmp2, a = _tmp1, c = _tmp2)
```

Placeholders are **not** holes (see below). An expression with placeholders like `_a_ * b + _c_` must *not* appear outside of a `functionof(...)` or `lawof(...)`.

Scoping rule. The scope of a placeholder is the nearest enclosing `functionof` or `lawof`. The same placeholder name may appear in different scopes without conflict:

```
functionof(functionof(_a_ * b, a = _a_)(some_value) + _a_, a = _a_)
```

A placeholder in an inner `functionof` or `lawof` **must** be bound there, so this code is invalid:

```
# DISALLOWED:
functionof(functionof(_a_ * b + _c_, a = _a_)(some_value) + _d_, c = _c_, d =
_d_)
```

4.11.2 Holes and `fn`

The reserved name _ denotes a **hole** — a position in a deterministic expression where an argument is not yet supplied. Holes are only valid inside the special operation `fn(...)`, which delimits the scope of hole lowering. The form `fn(expr)` wraps a hole expression and produces an anonymous function whose parameters are the holes in `expr`, in strict left-to-right reading order. This is analogous to the $f(\cdot, b)$ notation used in mathematics to denote a function with a free argument.

Each _ introduces a distinct positional parameter, named `arg1`, `arg2`, ... in left-to-right reading order. These names are normative and may be used as keyword arguments. Holes do not inherit keyword names from enclosing call positions.

Note: Holes work differently than placeholders (see above).

A single hole, resulting in a one-argument function:

```
neg = fn(0 - _)
poly = fn(polynomial(coefficients = cs, x = _))
```

The trivial case `fn(_)` is the identity function, equivalent to the built-in `identity`.

Multiple holes — left-to-right positional order:

```
g = fn(f(_, b, _))
h = fn(pow(_ / _, 2))
```

Each _ is distinct: `fn(_ * _)` multiplies two different inputs rather than squaring one. Use placeholders if arguments need to appear in the expression more than once, e.g. `functionof(_x_ * _x_, x = _x_)`.

Lowering. `fn(expr)` lowers to a `functionof` with placeholder variables. For example

```
g = fn(f(_, b, _))
```

lowers to

```
g = functionof(f(_arg1_, b, _arg2_), arg1 = _arg1_, arg2 = _arg2_)
```

which in turn lowers to

```
_tmp1 = elementof(anything)
_tmp2 = elementof(anything)
g = functionof(f(_tmp1, b, _tmp2), arg1 = _tmp1, arg2 = _tmp2)
```

_ may **not** appear on the left-hand side of a variable binding, and may **not** appear outside of `fn(...)`.

4.12 Higher-order operations

Broadcasting. `broadcast(f_or_K, name = array, ...)` or `broadcast(f_or_K, array, ...)` maps a function or kernel elementwise over arrays (and row-wise over tables; see tables). Keyword arguments bind inputs by name. If the callable has a declared positional order, positional binding is also permitted.

Deterministic broadcast with a named function:

```
b = 2 * a + 1
f = functionof(b, a = a)
C = broadcast(f, a = A)
```

With positional argument binding:

```
C = broadcast(f, A)
```

Using an anonymous function:

```
C = broadcast(fn(2 * _ + 1), A)
```

Multi-input broadcast:

```
d = a * x + b_param
g = functionof(d)
E = broadcast(g, a = slopes, x = points, b_param = intercepts)
```

Stochastic broadcast — kernel over array, producing an array-valued measure:

```
K = Normal(mu = a, sigma = 0.1)
D = draw(broadcast(K, a = A))
```

Return type:

- `broadcast(function, ...)` returns an **array value**.
- `broadcast(kernel, ...)` returns an **array-valued measure**: the independent product measure of the kernel applications at each array position.

The stochastic case returns a single product measure, not an array of measures. This respects the rule that measures are not stored inside arrays or records while still enabling vectorized stochastic model building.

Independence is explicit: Kernel broadcast means independent elementwise lifting. It does not cover dependent sequential kernels, autoregressive chains, or coupled array structure. For those, use `jointchain` or `chain` with explicit indexing.

Shape adaptation: `broadcast` follows standard broadcasting rules: singleton dimensions (length 1) are implicitly expanded to match the corresponding dimension of the other arguments. Scalar arguments are treated as arrays with all-singleton shape. Non-singleton dimensions must match across all arguments.

Tuple-returning callables: if `f` returns a tuple, `broadcast(f, ...)` returns a tuple of arrays (componentwise), not an array of tuples.

`broadcasted(f)` returns a callable that is equivalent to applying `broadcast` to `f` — that is, `broadcasted(f)(args...)` is equivalent to `broadcast(f, args...)`.

`reduce(f, xs)` is a fold over `xs` using the binary function `f`. For a vector `xs = [x0, x1, ..., xn-1]`, it computes `f(...f(f(x0, x1), x2)..., xn-1)`. For a table, `xs` is traversed row-wise and `f` takes two records (the accumulator and the next row). The first element (or row) is used as the initial accumulator value. Implementations may evaluate in parallel when `f` is associative. Unlike `broadcast`, `reduce` accepts only deterministic functions, not kernels.

`scan(f, init, xs)` is a left scan over `xs` using the binary function `f` with explicit initial accumulator `init`. It produces a vector (or table) of intermediate accumulator values, one per element/row of `xs`, of the same length as the input. Like `reduce`, `scan` accepts only deterministic functions.

4.13 Lowered linear form

A FlatPPL module admits a stable lowering to a linear SSA-style core form in which every non-atomic subexpression is bound to an auto-generated unique name (see Placeholders and holes for

the lowering stages). In the resulting form, every binding's right-hand side is either a literal or a function call: `name = c` or `name = f(name, ...)`. Operators, indexing, field access, and array literals all desugar to function calls (`add`, `get`, `vector`, etc.), giving the core form a uniform shape. This is a semantic property of FlatPPL modules; it is independent of the surface syntax.

4.14 Module composition

A FlatPPL module is a flat namespace of named bindings (see above). Modules can load other modules though, to make models composable:

`load_module(source)` loads a FlatPPL file and returns a module reference.

`source` may be a file path or a URL.

In the canonical syntax, bound names in the loaded module are accessed via dot syntax:

```
sig_module = load_module("signal.flatppl")
bkg_module = load_module("background.flatppl")

sig_model = sig_module.model
bkg_model = bkg_module.model
```

Access to loaded modules is not transitive: The loading module may access names in the loaded module, but not names in modules loaded by that module, so `sig_module.model` is valid but `sig_module.some_other_module.some_name` is not.

Load-time substitution. `load_module` may also be called with keyword arguments to substitute explicit input nodes of the loaded module:

```
sig_module = load_module("signal_channel.flatppl", mu = signal_strength, theta
= nuisance)
```

The left-hand side of each keyword argument must refer to an input of the loaded module. The phase of this input determines what it can be bound to on the right-hand side:

- `external` inputs of the loaded module may only be bound to **fixed** values in the loading module.
- `elementof` inputs of the loaded module may only be bound to **parameterized** values in the loading module.
- No other kinds of nodes in the loaded module may be bound to nodes in the loading module.

Value sets must be compatible in both cases, so the computational structure of the loaded module is not modified.

Path resolution. Relative file paths in `load_module(...)` are resolved relative to the directory of the FlatPPL file containing that `load_module(...)` call, not the host process's working directory. For embedded FlatPPL code, relative paths are resolved relative to the directory of the source file containing the embedded FlatPPL code block. The forward slash `/` is the mandatory path separator on all platforms. Parent-directory traversal via `..` is allowed. Absolute file paths are permitted but discouraged, as they prevent relocatable model repositories.

Aliasing is just assignment: `sig_model = sig_module.model` creates a local alias — a reference to the same underlying object in the loaded module's DAG, not a clone.

4.15 FlatPPL version compatibility

A FlatPPL module may declare which versions of FlatPPL it is compatible with via the reserved binding `flatppl_compat`:

```
flatppl_compat = "0.1"
```

The value is a string following Julia-style semantic versioning conventions: for pre-1.0 versions, the minor version is breaking (`"0.1"` means $\geq$ 0.1.0, $<$ 0.2.0); for versions $\geq$ 1.0, the major version is breaking (`"1"` means $\geq$ 1.0.0, $<$ 2.0.0). Multiple ranges are comma-separated and combined with OR:

```
flatppl_compat = "0.8, 0.9.2, 1.0.0, 2"
```

declares compatibility with versions v0.8.x, v0.9.2 upwards to v1, v1.x.y, and v2.x.y.

The declaration is optional. Short-lived models, didactic examples and the like may omit it. For embedded FlatPPL blocks, version compatibility may be managed at the host-language level (e.g. via Python or Julia package/environment dependency version bounds on FlatPPL packages). FlatPPL files of models intended for long-term use, publication or archival should definitely include a compatibility declaration.

The compatibility declaration of a loaded module is accessible via dot syntax (like any other bound value in the module): `some_module.flatppl_compat`.

---

5 Canonical syntax

This section specifies the canonical surface form of FlatPPL, used throughout this document as a notation for defining FlatPPL semantics and presenting examples. Note that the semantics of FlatPPL do not depend on this canonical syntax. Alternative syntactical representations may be advantageous for specific software ecosystems and use cases. They must map directly to and from the canonical syntax, though, with lossless round-trips except for formatting and whitespace.

5.1 Python/Julia-compatible syntax

The FlatPPL syntax is a subset of the intersection of valid (i.e. parsable) Python and Julia syntax.

FlatPPL code is therefore parseable by Python's `ast.parse()` and Julia's `Meta.parse()`, and no custom parser is required to implement FlatPPL engines in these languages. For other programming languages (e.g. C/C++), a standalone parser will be straightforward to implement, given the intentionally small grammar of FlatPPL.

Note that FlatPPL semantics are entirely different from both Python and Julia.

Embedding in host languages. The Python/Julia compatible AST design enables direct embedding of FlatPPL code as a domain-specific language (DSL). The host-language tooling parses the FlatPPL code, but it is then handed off to a FlatPPL engine as an AST, not interpreted or run as native host-language code.

In Python, FlatPPL can be embedded via a decorator:

```
@flatppl
def flatppl_module():
    mu = elementof(reals)
    a = draw(Normal(mu = mu, sigma = 1))
    m = lawof(a)
```

In **Julia**, via a macro:

```
flatppl_module = @flatppl begin
    mu = elementof(reals)
    a = draw(Normal(mu = mu, sigma = 1))
    m = lawof(a)
end
```

Note: These examples illustrate possible embedding approaches and are not normative; design choices regarding embedding are left to specific FlatPPL implementations.

5.2 Comments

Lines beginning with `#` (after optional whitespace) are comments and are ignored. Inline comments (`x = 3.14  # a comment`) are supported as well.

5.3 Supported constructs

FlatPPL has a very lean syntax:

- **Bindings**: `name = expr` and decomposition `a, b, c = expr` (see below).
- **Literals**: numbers (`3.14`, `42`), strings (`"foo"`), booleans (`true`, `false`), arrays (`[1, 2, 3]`), records (`record(a = 1, b = 2)`), tuples (`(a, b)`).
- **Infix arithmetic and comparisons**: `+`, `-`, `*`, `/`, unary `-`, `<`, `>`, `==`, `!=`, `<=`, `>=`.
- **Function calls**: `f(x, y)` (positional), `f(a = x, b = y)` (keyword) and `f(object, a = x, b = y)` (for some special operations).
- **Indexing and field access**: `A[i]`, `A[i, j]`, `A[:, j]`, `r.field`.

Also see the formal grammar below.

See binding names for rules on binding names.

5.4 Excluded constructs

The above are the only syntactical constructs allowed in FlatPPL. The following Python and Julia constructs, for example, are not allowed directly in canonical FlatPPL, but can easily be represented in other ways:

- **No ~ operator.** Use `draw()` instead.
- **No ** or ^ for exponentiation.** Use `pow(a, b)`.
- **No logical operators** (`and`/`or`/`not` in Python, `&&`/`||`/`!` in Julia). Use the functions `land`, `lor`, `lnot`, `lxor`.
- **No type annotations.** Types are inferred from the semantic rules.
- **No loops or conditionals.** Use `ifelse(cond, a, b)` for piecewise definitions (see logic and conditionals).
- **No function definition blocks.** Use `functionof` (see language design).
- **No implicit elementwise operators.** Infix `+`, `-`, `*`, `/` are not implicitly elementwise on arrays or matrices. Use `broadcast` (see broadcasting). This keeps matrix algebra unambiguous.

5.5 Decomposition syntax

The left side of an assignment may decompose an array, record, or tuple into named components:

```
a, b, c = draw(MvNormal(mu = mean_vector, cov = cov_matrix))
x, y = some_record
l, m, n = some_tuple
```

Decomposition is by position. For records, the field order determines which value each name receives; for arrays and tuples, positional index does. This is syntactic sugar: it lowers to an assignment followed by indexed or field-access bindings.

5.6 Indexing and slicing

FlatPPL uses **1-based indexing**.

`A[:, j]` selects all elements along the first axis at fixed index `j`. This lowers to `get(A, all, j)`, where `all` is a predefined selector meaning "entire axis."

```
A[:, j]          # → get(A, all, j)
A[i, :]          # → get(A, i, all)
T[:, :, k]       # → get(T, all, all, k)
T[i, :, k]       # → get(T, i, all, k)
```

5.7 Special operations

`elementof(S)`, `valueset(x)`, `draw(M)`, `lawof(...)`, `functionof(...)`, and `fn(...)` are special operations with their own syntax rules — they are not ordinary function calls. Their semantics are defined in language design. `load_module(...)` is documented in multi-file models.

5.8 Formal grammar

The canonical surface syntax is defined in EBNF below (ISO 14977-style, with `::=` for production and `|` for alternation). Operator precedence is encoded through stratified non-terminals (`Comparison` > `Additive` > `Multiplicative` > `Unary`).

```
(* Top level *)
Module          ::= Statement*
Statement       ::= Binding | Decomposition

(* Bindings *)
Binding         ::= Name "=" Expression
Decomposition   ::= Name ("," Name)+ "=" Expression

(* Expressions *)
Expression      ::= Comparison
Comparison      ::= Additive (CompOp Additive)?
Additive        ::= Multiplicative (AddOp Multiplicative)*
Multiplicative  ::= Unary (MulOp Unary)*
Unary           ::= "-" Unary | Postfix
Postfix         ::= Primary (FieldAccess | Indexing)*
Primary         ::= Literal | Name | Call | "(" Expression ")"

FieldAccess     ::= "." Name
Indexing        ::= "[" IndexExpr ("," IndexExpr)* "]"
IndexExpr       ::= Expression | ":"

CompOp          ::= "<" | ">" | "==" | "!=" | "<=" | ">="
AddOp           ::= "+" | "-"
MulOp           ::= "*" | "/"

(* Calls *)
Call            ::= Name "(" CallArgs? ")"
CallArgs        ::= PositionalArgs | KeywordArgs | MixedArgs
PositionalArgs  ::= Expression ("," Expression)*
KeywordArgs     ::= KeywordArg ("," KeywordArg)*
KeywordArg      ::= Name "=" Expression
MixedArgs       ::= LeadingPositional ("," KeywordArg)+
LeadingPositional ::= Expression

(* Literals *)
Literal         ::= Number | String | Boolean | ArrayLiteral | TupleLiteral
Number          ::= IntegerLit | RealLit
Boolean         ::= "true" | "false"
ArrayLiteral    ::= "[" (Expression ("," Expression)* ","?)? "]"
TupleLiteral    ::= "(" Expression "," Expression ("," Expression)* ","? ")"

(* Lexical *)
Name            ::= (Letter | "_") (Letter | Digit | "_")*
IntegerLit      ::= Digit+
RealLit         ::= Digit+ "." Digit* Exponent?
                  | Digit+ Exponent
                  | "." Digit+ Exponent?
Exponent        ::= ("e" | "E") ("+" | "-")? Digit+
```

```
String          ::= '"' StringChar* '"'
StringChar      ::= any character except '"' and '\' | '\' EscapeChar
Letter          ::= "a" .. "z" | "A" .. "Z"
Digit           ::= "0" .. "9"

(* Comments (treated as whitespace) *)
Comment         ::= "#" { any character except newline }
```

Note on `MixedArgs`. Syntactically, any `Call` may use `MixedArgs` (leading positional expression followed by keyword arguments). Semantically, only the special forms `functionof`, `lawof`, `broadcast`, and `load_module` accept this shape; other callables must use `PositionalArgs` or `KeywordArgs` only.

---

6 Measure algebra and analysis

This section documents the measure-level operations that form the compositional core of FlatPPL.

6.1 Measure-theoretic foundations

A **measurable space** is a pair (X, Σ_X) of a set and a σ-algebra. All spaces arising in FlatPPL are standard Borel spaces ($\mathbb{R}$, $\mathbb{Z}$, and finite products thereof), where the σ-algebra is the standard Borel σ-algebra and can be left implicit. A **measure** on X is a σ-additive function $\mu : \Sigma_X \to [0, \infty]$. A **probability measure** has $\mu(X) = 1$. All measures in FlatPPL are **σ-finite** (admitting a countable cover of finite-measure sets), which ensures that the Radon-Nikodym theorem applies (so densities exist with respect to a dominating reference measure) and that product and marginalization operations are well-defined. In the rest of this document, "measure" means "σ-finite measure."

A **transition kernel** (or **kernel**) from X to Y is a measurable function $\kappa : X \to M(Y)$, where $M(Y)$ is the space of measures on Y. When each $\kappa(x, \cdot)$ is a probability measure, the kernel is called a **Markov kernel**. In FlatPPL, kernels are represented as functions that map value points to measures.

The classical Giry monad operates on probability measures, which are normalized. FlatPPL extends this to σ-finite measures in general, e.g. to represent non-normalized posteriors and intensity measures. Staton (2017) provides the formal basis for this extension using the more general class of s-finite measures; all σ-finite measures are s-finite, so FlatPPL's algebraic operations are well-founded within that framework.

Density convention. All density formulas in this section are with respect to a reference measure implied by the constituent distribution types: Lebesgue for continuous variates, counting measure for discrete variates. When a kernel $\kappa(\theta)$ is parameterized by θ, the family is assumed dominated by a single θ-independent reference measure.

Normalization convention. Normalization is always explicit in FlatPPL. Built-in distribution/measure constructors do not normalize their inputs, and measure-algebra operations never rescale their inputs or outputs.

6.2 The measure monad

The Giry-style measure monad is defined by two operations:

- **Unit**: $\eta_X(x) = \delta_x$ (Dirac measure at x). In FlatPPL: `Dirac(value = v)`.
- **Bind**: $(\nu \gg\!= \kappa)(B) = \int_X \kappa(x)(B)\, d\nu(x)$. In FlatPPL: `chain(M, K)`.

6.3 Fundamental measures and measure algebra

6.3.1 Fundamental measures

FlatPPL provides three fundamental measures: the reference measures `Lebesgue` and `Counting`, and the point-mass measure `Dirac`.

- `Lebesgue(support = S)` — the canonical continuous reference measure on the support set `S`, restricted to `S`. For full-dimensional subsets of Euclidean or product spaces this is the ordinary Lebesgue measure on the ambient space. For lower-dimensional embedded affine sets such as `stdsimplex(n)`, it is the intrinsic affine Lebesgue measure on that set.

  `S` may be any FlatPPL set: one-dimensional (e.g. `reals`, `interval(0, 1)`, `posreals`), a Cartesian power (e.g. `cartpow(reals, n)`), a record-structured product (e.g. `cartprod(a = reals, b = posreals)`), a lower-dimensional embedded set (e.g. `stdsimplex(n)`) and so on (see sets).

  `iid(Lebesgue(reals), n)` is equivalent to `Lebesgue(cartpow(reals, n))`.

- `Counting(support = S)` — the counting measure on $\mathbb{Z}$, restricted to support `S`. Mass 1 at every integer in `S`. Reference measure for all discrete distributions.

- `Dirac(value = v)` — point-mass probability measure at `v` for any variate type.

The predefined constants `reals` (equivalent to `interval(-inf, inf)`) and `integers` (the set of all integers) serve as the default supports for the Lebesgue and counting measures respectively. The `support` parameter specifies where the measure is nonzero; density is zero outside. Measure algebra operations require their operands to share the same variate space (same type and dimension).

Uniform kernel extension. Mathematically, a measure is equivalent to a transition kernel with an empty first argument. So in FlatPPL, we unify measures and kernels and identify measures with nullary kernels. Measure algebra operations accept both kernels in general and measures as a (very important) special case of kernels. On a kernel, the operation applies to the output measure at each input point:

- `pushfwd(f, K)` denotes $\theta \mapsto \mathrm{pushfwd}(f, \kappa(\theta))$
- `weighted(w, K)` denotes $\theta \mapsto \mathrm{weighted}(w(\theta), \kappa(\theta))$

This applies to all measure-to-measure operations except `jointchain` and `chain`, which require non-nullary kernels in all but the first argument (see dependent composition).

Operations that map measures to values, like `totalmass`, `densityof`, and `logdensityof`, require closed measures (i.e. nullary kernels) as inputs. `densityof(M, x)` and `logdensityof(M, x)` evaluate the density of a measure at a point with respect to an implicit reference measure.

6.3.2 Density reweighting

- **`weighted(weight, base)`** — produces the measure $\nu(A) = \int_A f(x)\, dM(x)$, with $d\nu = f \cdot dM$, where f is the weight and M the base measure. The weight must be non-negative (constant or function). `normalize(weighted(f, Lebesgue(support = S)))` produces a probability distribution whose density w.r.t. Lebesgue on S is proportional to f.
- **`logweighted(logweight, base)`** — like `weighted`, but the weight is given in log-space: $d\nu = \exp(g) \cdot dM$.
- **`bayesupdate(L, prior)`** — reweights a prior measure by a likelihood object, producing the unnormalized posterior: $d\nu(\theta) = L(\theta) \cdot d\pi(\theta)$. Lowers to `logweighted(fn(logdensityof(L, _)), prior)`. See posterior construction for details.

6.3.3 Normalization and mass

- **`normalize(M)`** — given a measure M with finite total mass $Z = \text{totalmass}(M) > 0$, returns the probability measure M/Z. If $Z = 0$ or $Z = \infty$, the result is undefined. On a non-nullary kernel, normalizes the output measures.
- **`totalmass(M)`** — returns the total mass $Z = \int dM(x)$ as a scalar value. Requires a closed measure (not a non-nullary kernel).

6.3.4 Additive superposition

- **`superpose(M1, M2, ...)`** — measure addition: $\nu(A) = M_1(A) + M_2(A) + \ldots$ All components must share the same variate space. The result is generally not normalized. For example:

```
intensity = superpose(weighted(amplitude, signal_shape), bkg_shape)
events = draw(PoissonProcess(intensity = intensity))
```

  To build a normalized mixture distribution, use `normalize(superpose(weighted(w1, M1), weighted(w2, M2)))`. For example:

```
mix = normalize(superpose(weighted(a1, normal1), weighted(a2, normal2)))
```

6.3.5 Independent composition

- **`joint(M1, M2, ...)`** — independent product measure: $(M_1 \otimes M_2)(A \times B) = M_1(A) \cdot M_2(B)$.

  The output variate is formed by combining the component variates via `cat` (see array operations). All components must have the same shape class: all scalars (yielding an array), all arrays (yielding a concatenated array), or all records with distinct field names (yielding a merged record). Mixing shape classes is a static error.

For example, the measure product of a normal and an exponential probability measure, defined over a space of vectors, would be

```
M1 = Normal(mu = 0, sigma = 1)
M2 = Exponential(rate = 1.0)
vj = joint(M1, M2)
```

Keyword form. `joint(name1 = M1, name2 = M2, ...)` names the component variates, producing a measure over a space of records:

```
rj = joint(name1 = M1, name2 = M2)
```

is equivalent to `joint(relabel(M1, ["name1"]), relabel(M2, ["name2"]))`.

For kernels, `joint(K1, K2, ...)` results in a kernel that fans a single input out to all component kernels, so each of them receives the same input.

- **`iid(M, m, n, ...)`** — the product measure $M^{\otimes(m \cdot n \cdot \cdots)}$, producing a measure on arrays of shape `m × n × ...`.

  For example, to represent the draw of 100 IID samples from a normal distribution, use

```
obs = draw(iid(Normal(mu = a, sigma = b), 100))
```

6.3.6 Dependent composition

- **`chain(M, K1, K2, ...)`** — left-associative Kleisli composition (monadic bind). Keeps only the last kernel's variates, marginalizing out all intermediate variates. In contrast to standard Kleisli composition, the first argument may also be a measure (a nullary kernel). See `jointchain` below for the variant that retains all variates.

  Mathematically, we define the chain of a measure $\mu(A)$ and a transition kernel κ as

$$\nu(B) = \int \kappa(a, B)\, d\mu(a)$$

  This involves a marginalization integral, which is generally intractable. Left-associative.

```
prior_predictive = chain(prior, forward_kernel)
```

Equivalence with stochastic nodes:

```
model = chain(M1, K2, K3)
```

is equivalent to

```
a = draw(M1)
b = draw(K2(a))
c = draw(K3([a, b]))
model = lawof(c)
```

- **`jointchain(M, K1, K2, ...)`** — dependent joint measure. The first argument is a base measure or kernel; the remaining arguments are non-nullary kernels whose inputs bind to the variates of everything to their left.

  `jointchain` is left-associative. In contrast to `chain`, the output variate is the `cat` of the variates of all the components, as with `joint`.

  Keyword form. `jointchain(name1 = M, name2 = K1, ...)` names the component variates, producing a measure over a space of records. It is equivalent to `jointchain(relabel(M, ["name1"]), relabel(K1, ["name2"]), ...)`.

  Mathematically, we define the joint chain of a measure $\mu(A)$ and a transition kernel κ as

  $$\nu(A \times B) = \int_A \kappa(a, B)\, d\mu(a)$$

  The density of the joint chain is the product of the constituent conditional densities — no marginalization integral is involved, unlike with `chain`. So density is tractable if the densities of all the components are.

  Equivalence with stochastic nodes:

  ```
  model = jointchain(M1, K2, K3)
  ```

  is equivalent to

  ```
  a = draw(M1)
  b = draw(K2(a))
  c = draw(K3([a, b]))
  model = lawof([a, b, c])
  ```

  Relationship to `chain`:

  ```
  jointchain(M, K)
  ```

  is equivalent to

  ```
  chain(M, functionof(joint(Dirac(value = _a_), K(_a_)), a = _a_))
  ```

6.3.7 Support restriction

- **`truncate(M, S)`** — restricts the support of measure `M` to the set `S`: $\nu(A) = M(A \cap S)$. Does not normalize automatically.

```
half_normal = normalize(truncate(Normal(mu = 0, sigma = 1), interval(0,
inf)))
```

6.3.8 Transformation and projection

- **`pushfwd(f, M)`** — pushforward of measure M through function f:

$$(f_* M)(Y) = M(f^{-1}(Y))$$

  For kernels, `pushfwd` acts on their output measures.

  For example, a log-normal probability measure can be constructed as

```
mu = Normal(mu = 0, sigma = 1)
nu = pushfwd(exp, mu)  # → LogNormal
```

  The equivalent in stochastic-node form is:

```
mu = Normal(mu = 0, sigma = 1)
x = draw(mu)
y = exp(x)
nu = lawof(y)
```

  A pushforward can also be used to project, respectively marginalize:

```
mu = relabel(iid(Normal(mu = 0, sigma = 1), 3), ["a", "b", "c"])
pushfwd(fn(get(_, ["a", "c"])), model)   # marginalizes out b
```

- **`bijection(f, f_inv, logvolume)`** annotates a function `f` with its inverse `f_inv` and the log-volume-element `logvolume` of the forward map. The result is a function that is semantically `f`.

  FlatPPL engines will often need the inverse of `f` and the volume element when computing densities of pushforward measures. Function inverses are hard to derive automatically and the computation of Jacobian determinant via automatic differentiation can be very inefficient, while the user or system that authors/generates FlatPPL may have access to both in closed form.

  `logvolume` is the generalized log-volume-element of the forward function — it generalizes the log-absolute-determinant of the Jacobian to mappings between spaces of different dimension. It may be a function or a scalar value (`logvolume = 0` for volume-preserving bijections). The convention is that `logvolume` describes the forward map.

  The user asserts that `f_inv` is the inverse of `f` and that `logvolume` is correct with respect to how `f` is used in the FlatPPL module. FlatPPL implementations are not required to verify this.

For standard cases like `exp`, FlatPPL engines can be expected to know the inverse and volume element, but it would be written in FlatPPL as

```
exp_bijection = bijection(exp, log, identity)
```

A more interesting example that includes an explicit definition of domain and codomain of the function is squaring on the positive reals:

```
pos_x = elementof(interval(0, inf))
sq = bijection(
    functionof(pow(pos_x, 2), x = pos_x),
    functionof(sqrt(pos_x), x = pos_x),
    fn(log(2 * _))
)
```

6.4 Likelihoods and posteriors

6.4.1 Likelihood construction

`likelihoodof(K, obs)` takes a kernel `K` and observed data `obs`, and produces a **likelihood object**: the density of `K` evaluated at `obs`, as a function of the kernel's input parameters. The result is a semantic object, not a plain function — this prevents accidental confusion between density and log-density values. Likelihood values are extracted explicitly via `densityof(L, theta)` and `logdensityof(L, theta)`.

Mathematically, `densityof(likelihoodof(K, obs), theta)` corresponds to $\mathrm{pdf}(\kappa(\theta), x)$, where κ is the kernel and x the observed data.

Multiple observations. `likelihoodof` does not implicitly construct IID products of the model kernel. The shape of variates of (the probability measures generated by) the kernel must match the shape of the observed data. Product kernels must be created explicitly, e.g. via `iid` for multiple IID observations.

Region-restricted likelihoods are constructed by explicitly restricting the model and filtering the data, based on a validity region (represented by a set) before constructing the likelihood.

For IID observation models with `n` observations:

```
R = interval(-3.0, 3.0)
obs_R = filter(fn(_ in R), obs_values)
n = length(obs_R)
model_R = normalize(truncate(model, R))
L_R = likelihoodof(iid(model_R, n), obs_R)
```

For Poisson process models (note that `truncate` does not normalize, this is important here):

```
R = interval(-3.0, 3.0)
obs_R = filter(fn(_ in R), obs_events)
model_R = PoissonProcess(intensity = truncate(intensity, R))
L_R = likelihoodof(model_R, obs_R)
```

For binned count models, use `selectbins` to select whole bins:

```
R = interval(-3.0, 3.0)
obs_R = selectbins(edges, R, obs_counts)
expected_R = selectbins(edges, R, expected_counts)
model_R = broadcast(fn(Poisson(rate = _)), expected_R)
L_R = likelihoodof(model_R, obs_R)
```

6.4.2 Combining likelihoods

`joint_likelihood(L1, L2, ...)` combines multiple likelihoods into a single likelihood by multiplying their density values (equivalently, summing log-densities):

$$\log L(\theta) = \log L_1(\theta) + \log L_2(\theta) + \ldots$$

A joint likelihood

```
L1 = likelihoodof(model1, obs1)
L2 = likelihoodof(model2, obs2)
L = joint_likelihood(L1, L2)
```

is equivalent to

```
model = joint(model1, model2)
obs = cat(obs1, obs2)
L = likelihoodof(model, obs)
```

6.4.3 Posterior construction

`bayesupdate(L, prior)` produces the **unnormalized** posterior measure:

$$\nu(A) = \int_A L(\theta)\, d\pi(\theta)$$

with density

$$d\nu(\theta) = L(\theta) \cdot d\pi(\theta)$$

For example

```
mu = elementof(reals)
model = Normal(mu = mu, sigma = 1.0)
obs = 2.5
```

```
L = likelihoodof(model, obs)
prior = joint(mu = Normal(mu = 0, sigma = 2.0))
posterior = bayesupdate(L, prior)
```

`bayesupdate` can be lowered to `logweighted`:

```
pstr = bayesupdate(L, prior)
```

is equivalent to

```
pstr = logweighted(fn(logdensityof(L, _)), prior)
```

The evidence `Z` can be expressed as

```
Z = totalmass(pstr)
```

though it is typically not tractable.

6.4.4 Structural disintegration

Bayesian models are sometimes expressed by direct construction of the joint probability measure over parameters and observations. Stan-like probabilistic languages primarily or exclusively express Bayesian models this way. To construct a FlatPPL likelihood and posterior from such a joint model, the joint must be split into a forward kernel (observation model), and a marginal measure (prior). The forward kernel can then be combined with some observed data to build a likelihood.

In measure theory, such a decomposition is known as disintegration. Given a space of parameters $\mathcal{A}$ and a space of observations $\mathcal{B}$, and a joint measure μ on the joint measurable space $\mathcal{A} \times \mathcal{B}$, the disintegration theorem states that (for standard Borel spaces, which all FlatPPL spaces are) there exists a kernel $\kappa : \mathcal{A} \to M(\mathcal{B})$ and a marginal measure ν on $\mathcal{A}$ such that:

$$\mu(A \times B) = \int_A \kappa(a, B)\, d\nu(a)$$

This is the generalization of conditional probability to arbitrary measures.

FlatPPL does not support arbitrary disintegration (the general theorem allows for disintegration along arbitrary measurable functions, not just orthogonal projections), but it does support **structural disintegration** via `disintegrate`, which returns the kernel κ and the marginal ν together as a tuple. It decomposes the DAG of a joint measure, given the names or indices of the joint variates that correspond to the variates of the forward kernel (and so also correspond to the entries of the observed data).

For example:

```
# Equivalent to a Stan/Pyro/Turing.jl model
sigma = 1.0
a = draw(Normal(mu = 0.0, sigma = 2.0))
b = draw(Normal(mu = a, sigma = sigma))
joint_model = lawof(record(a = a, b = b))

# Structural disintegration
forward_kernel, prior = disintegrate(["b"], joint_model)

# Now construct likelihood and posterior
obs = record(b = 2.1)
L = likelihoodof(forward_kernel, obs)
posterior = bayesupdate(L, prior)
```

`disintegrate(selector, joint_measure)` returns a tuple `(kernel, base_measure)`, where `kernel` is the conditional kernel for the selected variates and `base_measure` is the marginal base measure — the measure obtained by marginalizing the selected variates out of the joint.

Selectors work like in `get`: `"b"` selects the bare value, `["b"]` selects a `record(b = ...)`.

`kernel, base_measure = disintegrate(selector, joint_measure)` must satisfy the condition that `jointchain(base_measure, kernel)` is equivalent to `joint_measure`.

For the large class of joint models whose factorization structure is explicit in the DAG, `disintegrate` can be implemented via straightforward graph inspection. For models that involve internal marginalization, non-bijective changes of variables, or other transformations that destroy explicit factorization structure, the decomposition may be intractable and may not be supported.

---

7 Built-in functions

This section provides reference documentation for all deterministic functions and value-level operations in FlatPPL. For measure-level operations, see measure algebra and analysis. For distribution constructors, see built-in distributions.

7.1 Identities

- **`identity(x)`** — the identity function: returns its argument unchanged. Equivalent to `fn(_)`.

7.2 Array and table generation

- **`vector(x1, x2, ...)`** — constructs a 1D array (vector) from the given elements. Equivalent to the array literal syntax `[x1, x2, ...]`.
- **`fill(x, n, m, ...)`** — creates an array of shape `n × m × ...` filled with value `x` (e.g., `fill(0.0, 10)`).
- **`zeros(n, m, ...)`** — creates a real-valued array of shape `n × m × ...` filled with zeros. Equivalent to `fill(0, n, m, ...)`.

- **`ones(n, m, ...)`** — creates a real-valued array of shape `n × m × ...` filled with ones. Equivalent to `fill(1, n, m, ...)`.
- **`eye(n)`** — creates the $n \times n$ identity matrix I_n.
- **`onehot(i, n)`** — length-n basis vector e_i with one at position i and zero elsewhere, for $i \in \{1, \ldots, n\}$.
- **`linspace(from, to, n)`** — returns an endpoint-inclusive range of `n` real numbers, evenly spaced from `from` to `to` (both included). The range is semantically a vector of reals.

  ```
  linspace(0.0, 10.0, 5)     # equivalent to [0.0, 2.5, 5.0, 7.5, 10.0]
  ```

  Note: When used to specify a binning, `n` is the number of bin **edges** (producing n-1 bins).
- **`extlinspace(from, to, n)`** — extended `linspace` with overflow edges. Semantically equivalent to `cat([-inf], linspace(from, to, n), [inf])`, producing n+2 edge points and n+1 bins (n-1 finite bins plus 2 overflow bins).

  ```
  extlinspace(0.0, 10.0, 5)  # equivalent to [-inf, 0.0, 2.5, 5.0, 7.5, 10.0,
  inf]
  ```

  `extlinspace` provides a convenient way to define binnings with underflow and overflow bins without constructing explicit vectors. Note that in this case `n` specifies the number of finite edge points; `extlinspace(from, to, n)` produces `n + 2` total edge points (adding `-inf` and `inf`) and a total of `n + 1` bins (including the overflow bins).
- **`load_data(source, valueset)`** — loads a collection of data entries from an external source and returns a vector or table. The shape of the result is determined by the declared `valueset`, which defines the set that governs each vector entry or table row.
  - `source`: a file path or URL identifying the data source. File path resolution follows the same rules as with `load_module`.
  - `valueset`: specifies the set that governs each vector entry or table row.

  This loads a table with a scalar column `a` and a 3-vector column `b`:

  ```
  events = load_data(
      source = "observed_events.csv",
      valueset = cartprod(a = reals, b = cartpow(reals, 3)))
  ```

  This loads a flat vector of real values:

  ```
  weights = load_data(source = "weights.csv", valueset = reals)
  ```

  Tabular data with a single column can be loaded as a vector instead of a table, depending on `valueset`.

All FlatPPL engines must support at least:

- **JSON** (`.json`) — containing either an array of objects (array-of-structs), an object of arrays (struct-of-arrays) or a vector.
- **CSV and WSV** (`.csv`, `.wsv`) — comma- or whitespace-separated values with column names in the first row.
- **Arrow IPC** (`.arrow`, `.arrows`) — Apache Arrow File and Stream formats.

7.3 Field and element access

- **`get(container, selectors...)`** — unified element access and subset selection. `selectors` may be a single name or array of names, or a single or multiple integer indices, or arrays of integer indices. Tuples use a single integer literal index.

  Element access (single selection — returns a single element):

  ```
  get(r, "a")           # record element access
  get(v, 3)             # array element access
  get(v, 2, 3)          # multi-dimensional array element access
  get(t, 1)             # first tuple component (integer literal index, 1-
  based)
  ```

  Subset selection (multi-selection — returns a sub-container of the same kind):

  ```
  get(r, ["a", "c"])    # record subset selection
  get(A, [1, 3, 4], 2)  # array subset selection
  ```

  Surface syntax lowering: FlatPPL's indexing and field-access syntax lowers to `get`: `r.a` ≡ `get(r, "a")`, `v[i]` ≡ `get(v, i)`, `A[i, j]` ≡ `get(A, i, j)`.

  `get` with a subset selector and a hole expression produces a projection function. For example, `pushfwd(fn(get(_, ["a", "c"])), M)` marginalizes M over all fields except "a" and "c".

  Note: module member access via dot syntax (`sig.model` where `sig` is a loaded module) is a separate syntactic category — modules are namespace references, not record values, and module dot access does not lower to `get`.

  Axis slicing with `all`. For matrices and multi-dimensional arrays, the keyword `all` selects an entire axis: `get(M, i, all)` returns row i, `get(M, all, j)` returns column j. Surface syntax `M[:, j]` lowers to `get(M, all, j)`.

7.4 Array and table operations

`cat(x, y, ...)` concatenates values of the same structural kind:

- **`cat(vector1, vector2, ...)`** concatenates vectors.

  Example: `cat([1, 2, 3], [4, 5])` produces `[1, 2, 3, 4, 5]`.

- **`cat(record1, record2, ...)`** merges records, concatenating their field lists in order.

Example: `cat(record(a=1, b=2), record(c=3))` produces `record(a=1, b=2, c=3)`.

Duplicate field names across the input records are a static error. Concatenation of a mix of vectors and records is also not permitted.

`rowstack(vs)` constructs a matrix whose rows are the vectors in `vs`. The argument `vs` is a vector of vectors, all of the same length.

```
M = rowstack([[1, 2, 3], [4, 5, 6]])
```

returns

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

`colstack(vs)` constructs a matrix whose columns are the vectors in `vs`. The argument `vs` is a vector of vectors, all of the same length.

```
M = colstack([[1, 2, 3], [4, 5, 6]])
```

returns

$$M = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

`partition(xs, spec)` splits a vector `xs` into a vector of sub-vectors. The second argument `spec` may be:

- A positive integer `n`: split `xs` into equal groups of size `n`. Requires `length(xs)` to be divisible by `n`.
- A vector of positive integers `[n1, n2, ...]`: split `xs` into groups of the given sizes in order. Requires `sum([n1, n2, ...])` to equal `length(xs)`.

`partition(xs, n)` is equivalent to `partition(xs, fill(n, div(length(xs), n)))`.

For example:

```
partition([1, 2, 3, 4, 5, 6], 3)    # [[1, 2, 3], [4, 5, 6]]
partition([1, 2, 3, 4, 5], [2, 3])  # [[1, 2], [3, 4, 5]]
```

`reverse(xs)` reverses the order of elements in a vector or rows in a table.

7.5 Scalar restrictions and constructors

These functions set-restrict or construct scalar values (see value types for set definitions).

Function	Arguments	Description	Domains
`boolean`	`x`	returns `x` when `x in booleans`	any scalar numeric
`integer`	`x`	returns `x` when `x in integers`	any scalar numeric
`real`	`x`	returns `x` (or $\mathrm{Re}(x)$ for complex)	any scalar numeric
`complex`	`re`, `im`	$\mathrm{re} + i \cdot \mathrm{im}$	`reals`
`string`	`x`	returns `x`	`string`
`imag`	`x`	$\mathrm{Im}(x)$	`reals`, `complexes`

7.6 Elementary functions

The following standard mathematical functions are predefined. All accept scalar arguments and return scalar results. They have positional calling conventions with defined argument order.

Function	Arguments	Description	Domains
`exp`	`x`	e^x	`reals`, `complexes`
`log`	`x`	$\ln(x)$	`posreals`, `complexes`
`log10`	`x`	$\log_{10}(x)$	`posreals`
`pow`	`base`, `exponent`	$\mathrm{base}^{\mathrm{exponent}}$	`reals`, `complexes`
`sqrt`	`x`	$\sqrt{x}$	`nonnegreals`, `complexes`
`abs`	`x`	$\lvert x \rvert$	`reals`, `complexes`
`abs2`	`x`	$\lvert x \rvert^2$	`reals`, `complexes`
`sin`	`x`	$\sin(x)$	`reals`, `complexes`
`cos`	`x`	$\cos(x)$	`reals`, `complexes`
`min`	`a`, `b`	$\min(a, b)$	`reals`
`max`	`a`, `b`	$\max(a, b)$	`reals`

Function	Arguments	Description	Domains
`floor`	`x`	$\lfloor x \rfloor$	`reals`
`ceil`	`x`	$\lceil x \rceil$	`reals`
`round`	`x`	nearest integer, half to even (IEEE 754 default)	`reals`
`div`	`a`, `b`	$\lfloor a/b \rfloor$	`integers`, `b` $\neq 0$
`mod`	`a`, `b`	$a - b \cdot \lfloor a/b \rfloor$	`integers`, `b` $\neq 0$
`conj`	`x`	conjugate $\bar{x}$	`reals`, `complexes`
`cis`	`theta`	$e^{i\theta}$	`reals`

For complex arguments, `log` and `sqrt` use the principal branch ($\arg(z) \in (-\pi, \pi]$). `pow` extends via $z^w = e^{w \log z}$ (principal branch); either or both arguments may be complex.

7.7 Operator-equivalent functions

FlatPPL arithmetic operators cannot themselves be used as first-class function names. Instead, they lower to the following named function equivalents, which can also be be passed as arguments to higher-order functions like `broadcast`, `reduce` and `scan`.

Arithmetic functions:

Function	Arguments	Corresponds to	Domains
`add`	`a`, `b`	`a + b`	scalars or arrays of same shape (real or complex)
`sub`	`a`, `b`	`a - b`	scalars or arrays of same shape (real or complex)
`mul`	`a`, `b`	`a * b`	scalars; matrix/matrix and matrix/vector products
`divide`	`a`, `b`	`a / b`	scalars (real or complex)
`neg`	`x`	`-x`	scalars or arrays (real or complex)

Comparison functions:

Function	Arguments	Corresponds to	Domains
`equal`	a, b	$a = b$	`integers`, `booleans`, strings
`unequal`	a, b	$a \neq b$	`integers`, `booleans`, strings
`lt`	a, b	$a < b$	`reals`
`le`	a, b	$a \leq b$	`reals`
`gt`	a, b	$a > b$	`reals`
`ge`	a, b	$a \geq b$	`reals`

Exact equality (`equal` / `==` and `unequal` / `!=`) is restricted to discrete domains to avoid dependence on numerical precision. To compare real-valued quantities for exact equality, use a function that guarantees a discrete result like `integer(x)`, `floor(x)`, `ceil(x)`, or `round(x)`.

7.8 Scalar predicates

Function	Arguments	Description	Domains
`isfinite`	x	x is a finite number (not ±∞, not NaN)	`reals`, `complexes`
`isinf`	x	x is $+\infty$ or $-\infty$	`reals`, `complexes`
`isnan`	x	x is NaN	`reals`, `complexes`
`iszero`	x	x is exactly zero	`reals`, `integers`, `complexes`

`iszero(x)`, unlike `x == 0`, allows non-discrete inputs. `iszero` checks that its argumet is exactly zero, with no tolerance for numerical precision.

7.9 Checked values

`checked(value, condition)` is a value-preserving assertion: it returns `value` unchanged if `condition` evaluates to `true`, and raises a static error otherwise.

- `value` — any expression; `checked` returns it with identical type and phase.
- `condition` — must be a fixed-phase boolean, evaluated at load/inference time.

```
n_raw = external(integers)
data = load_data(source = "...", valueset = reals)
n = checked(value = n_raw, condition = equal(n_raw, length(data)))
# n is n_raw with the dimension check attached; use n downstream.
```

The canonical calling form uses keyword arguments; `checked(value_expr, condition = ...)` is also accepted. Because `checked` threads the value through to downstream use, the check is topologically tied to that use and cannot be eliminated by term-rewriting passes — ensuring the invariant is always validated.

7.10 Linear algebra

Function	Arguments	Description	Domains
`transpose`	`A`	A^T	matrices
`adjoint`	`A`	$A^\dagger$ (conj. transpose)	matrices
`det`	`A`	$\det(A)$	square matrices
`logabsdet`	`A`	$\log\lvert\det(A)\rvert$	square matrices
`inv`	`A`	A^{-1}	square matrices
`trace`	`A`	$\mathrm{tr}(A)$	square matrices
`linsolve`	`A`, `b`	solve $Ax = b$ for x	square `A`, vector `b`
`lower_cholesky`	`A`	triangular L with $A = LL^\dagger$	positive definite `A`
`row_gram`	`A`	$AA^\dagger$	matrices
`col_gram`	`A`	$A^\dagger A$	matrices
`self_outer`	`x`	$x \cdot x^\dagger$ (outer product)	vectors
`diagmat`	`x`	$\mathrm{diag}(x_1, ..., x_n)$	vectors

Matrix multiplication and addition use the standard * and + operators.

7.11 Reductions

Function	Arguments	Description	Domains
`sum`	array	sum of elements	real/complex arrays
`product`	array	product of elements	real/complex arrays
`maximum`	array	maximum of elements	real arrays
`minimum`	array	minimum of elements	real arrays
`length`	array/table	number of elements / rows	arrays, tables

7.12 Norms and normalization

Function	Arguments	Description	Domains
`l1norm`	`v`	$\sum_i \lvert v_i\rvert$	real/complex vectors
`l2norm`	`v`	$\sqrt{\sum_i \lvert v_i\rvert^2}$	real/complex vectors

Function	Arguments	Description	Domains
`l1unit`	`v`	$v/\|v\|_1$	real/complex vectors
`l2unit`	`v`	$v/\|v\|_2$	real/complex vectors
`logsumexp`	`v`	$\log \sum_i e^{v_i}$	real vectors
`softmax`	`v`	$(e^{v_i}/\sum_j e^{v_j})_i$	real vectors
`logsoftmax`	`v`	$(v_i - \log \sum_j e^{v_j})_i$	real vectors

7.13 Logic and conditionals

Function	Arguments	Description	Domains
`land`	`a`, `b`	logical conjunction	`booleans`
`lor`	`a`, `b`	logical disjunction	`booleans`
`lnot`	`a`	logical negation	`booleans`
`lxor`	`a`, `b`	logical exclusive-or	`booleans`
`ifelse`	`cond`, `a`, `b`	returns `a` if `cond` is true, `b` otherwise	`cond`: `booleans`; `a`, `b`: `anything`

7.14 Membership, filtering, and bin selection

- **`x in S`** — returns `true` if `x` lies in set `S`, else `false`. The type of `x` must match the element type of set `S`.
- **`filter(pred, data)`** — filters an array or table by a boolean predicate, returning a shorter array or table containing only elements/rows for which `pred` returns `true`.

```
data_in_range = filter(fn(_ in interval(2.0, 8.0)), data)
```

- **`selectbins(edges, region, counts)`** — selects whole-bin counts for bins whose intervals intersect `region`. Returns a shorter count array. No fractional-bin clipping or rebinning is applied, bins are either fully included or excluded.

```
restricted_counts = selectbins(edges, interval(2.0, 8.0), observed_counts)
```

7.15 Binning

- **`bincounts(bins, data)`** — counts data points falling into the given bins. Data points outside all bins are ignored.

  1D case: `bins` is a vector of bin edges (n+1 edges define n bins).

Bin edges may be explicit vectors or generated via `linspace` or `extlinspace`.

```
bincounts([0.0, 2.5, 5.0, 7.5, 10.0], data)  # 4 bins, explicit edges
bincounts(linspace(0.0, 10.0, 5), data)      # 4 bins, equivalent
bincounts(extlinspace(0.0, 10.0, 5), data)   # 6 bins (4 finite + 2
overflow)
```

Multi-dimensional case: `bins` is a record of edge vectors, one per field. The data must be a record of equally-sized arrays matching the field names. The result is a multi-dimensional array of counts whose axis order follows the field order of `bins`.

```
bincounts(
    record(a = linspace(100, 140, 5), b = linspace(0, 100, 4)),
    data
) # → array of size 4 x 3
```

Bin intervals. Given $n + 1$ edges $x_1, x_2, ..., x_{n+1}$, bins are left-closed and right-open $[x_i, x_{i+1})$ for $i \in \{1, ..., n - 1\}$, except for the last bin which is also closed on the right $[x_n, x_{n+1}]$. This ensures that a value exactly at the upper boundary falls into the last bin.

7.16 Interpolation functions

7.16.1 Three-point interpolation functions

FlatPPL provides five three-point interpolation functions that are general purpose but compatible with the interpolation methods used in RooFit, HistFactory, pyhf, and HS³ (see pyhf and HistFactory compatibility).

These interpolation functions are deterministic, value-level functions that interpolate between given anchor output values at $\alpha = -1$, $\alpha = 0$, and $\alpha = +1$ for a given $-\infty < \alpha < \infty$.

All of these functions share the same signature:

```
interp_*(left, center, right, alpha)
```

- `left`: anchor output value at $\alpha = -1$
- `center`: anchor output value at $\alpha = 0$
- `right`: anchor output value at $\alpha = +1$
- `alpha`: evaluation point.

Function	Interpolation	Extrapolation	HS³	pyhf
`interp_pwlin`	piecewise linear	continuation	`lin`	code0

Function	Interpolation	Extrapolation	HS³	pyhf
`interp_pwexp`	piecewise exponential	continuation	`log`	code1
`interp_poly2_lin`	quadratic	linear	`parabolic`	code2
`interp_poly6_lin`	6th-order polynomial	linear	`poly6`	code4p
`interp_poly6_exp`	6th-order polynomial	exponential	—	code4

`interp_poly6_exp` exists in pyhf (code4) but is not part of the HS³ standard yet.

`interp_pwlin(left, center, right, alpha)` — piecewise linear interpolation:

$$\text{For} \quad \alpha \geq 0: \quad f(\alpha) = \text{center} + \alpha \cdot (\text{right} - \text{center})$$

$$\text{For} \quad \alpha < 0: \quad f(\alpha) = \text{center} + \alpha \cdot (\text{center} - \text{left})$$

Non-differentiable at $\alpha = 0$ in general.

`interp_pwexp(left, center, right, alpha)` — `interp_pwlin` applied in log-space: equivalent to `exp(interp_pwlin(log(left), log(center), log(right), alpha))`. Requires strictly positive values for `left`, `center` and `right`. The result is always positive.

Non-differentiable at $\alpha = 0$ in general.

`interp_poly2_lin(left, center, right, alpha)` — quadratic interpolation inside $[-1, +1]$, linear extrapolation outside:

$$S = (\text{right} - \text{left})/2, \quad A = (\text{right} + \text{left})/2 - \text{center}$$

$$\text{For } |\alpha| \leq 1: \quad f(\alpha) = \text{center} + S \cdot \alpha + A \cdot \alpha^2$$

Outside $[-1, +1]$, the function continues linearly with slope $S + 2A$ (right) or $S - 2A$ (left).

`interp_poly6_lin(left, center, right, alpha)` — 6th-order polynomial inside $[-1, +1]$, linear extrapolation outside. The polynomial satisfies five constraints: $f(-1) = \text{left}$, $f(0) = \text{center}$, $f(+1) = \text{right}$, and C^1 continuity at $\alpha = \pm 1$ (matching the linear extrapolation slopes).

`interp_poly6_exp(left, center, right, alpha)` — 6th-order polynomial inside $[-1, +1]$, exponential extrapolation outside. For $|\alpha| > 1$:

$$f(\alpha) = f(\pm 1) \cdot \exp((\alpha \mp 1) \cdot f'(\pm 1)/f(\pm 1))$$

The polynomial coefficients differ from `interp_poly6_lin` because the derivative-matching conditions at $\alpha = \pm 1$ target the exponential slopes. The result stays positive, making this appropriate for multiplicative factors.

7.17 Approximation functions

`polynomial(coefficients, x)` — power-series polynomial $\sum a_i x^i$. Non-negativity over the intended support is the user's responsibility.

`bernstein(coefficients, x)` — Bernstein basis polynomial, guaranteed non-negative when all coefficients are non-negative. Defined on $[0, 1]$; the support interval of the surrounding `Lebesgue` provides the rescaling range.

`stepwise(edges, values, x)` — piecewise-constant step function. Strictly piecewise constant (no implicit interpolation). The length of vector `values` must be one less than the length of vector `edges`.

---

8 Built-in distributions

This section catalogs the built-in distributions (i.e. probability measures) provided by FlatPPL.

The distribution constructors listed here are FlatPPL Markov kernels and the distribution parameters are kernel inputs/arguments. The kernels follow the general calling conventions. The names and order of the distribution parameters specified below define the names and positional order of the kernel arguments.

Variate domain and support. The catalog below lists both variate domain and support for each distribution. The domain is the set over which density evaluation is defined (returning 0 outside the support). The support is the set where the density is nonzero. Samples always fall within the support.

Probability density and mass functions are given as densities in the Radon-Nikodym sense, for both continuous and discrete distributions. The reference measure is specified as well.

Note. Probability distributions with user-defined densities may be constructed compositionally via `normalize(weighted(f, Lebesgue(S)))` — see measure algebra for details.

8.1 Standard continuous distributions

Distribution	Parameters	Domain	Support
`Uniform`	`support`	`reals`	`support`
`Normal`	`mu`, `sigma`	`reals`	`reals`
`GeneralizedNormal`	`mean`, `alpha`, `beta`	`reals`	`reals`
`Cauchy`	`location`, `scale`	`reals`	`reals`
`StudentT`	`nu`	`reals`	`reals`
`Logistic`	`mu`, `s`	`reals`	`reals`
`LogNormal`	`mu`, `sigma`	`reals`	`posreals`

Distribution	Parameters	Domain	Support
`Exponential`	`rate`	`reals`	`nonnegreals`
`Gamma`	`shape`, `rate`	`reals`	`posreals`
`Weibull`	`shape`, `scale`	`reals`	`nonnegreals`
`InverseGamma`	`shape`, `scale`	`reals`	`posreals`
`Beta`	`alpha`, `beta`	`reals`	`unitinterval`

`Uniform(support)` — The uniform distribution on `support`.

Domain/Support: ambient value space of `support` / `support`.

Parameters:

- `support`: any FlatPPL set S with $0 < \lambda(S) < \infty$, where λ is `Lebesgue(S)`.

Density w.r.t. `Lebesgue(support = S)` inside of `S`:

$$\frac{1}{\lambda(S)},$$

where $\lambda = \mathrm{Lebesgue}(\mathrm{support} = S)$ is the canonical continuous reference measure associated with S.

`Uniform(S)` is equivalent to `normalize(Lebesgue(S))`.

`Normal(mu, sigma)` — The normal (or Gaussian) distribution.

Domain/Support: `reals`/`reals`.

Parameters:

- `mu = elementof(reals)`: the mean μ.
- `sigma = elementof(posreals)`: the standard deviation σ.

Density w.r.t. `Lebesgue(reals)`:

$$\frac{1}{\sigma\sqrt{2\pi}} \exp\!\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

`GeneralizedNormal(mean, alpha, beta)` — The symmetric generalized normal distribution. Reduces to the normal distribution when $\beta = 2$.

Domain/Support: `reals`/`reals`.

Parameters:

- `mean = elementof(reals)`: location μ.
- `alpha = elementof(posreals)`: scale.
- `beta = elementof(posreals)`: shape.

Density w.r.t. `Lebesgue(reals)`:

$$\frac{\beta}{2\alpha\,\Gamma(1/\beta)}\exp\left(-\left(\frac{|x-\mu|}{\alpha}\right)^{\beta}\right)$$

`Cauchy(location, scale)` — The Cauchy distribution. Equivalent to `StudentT(1, location, scale)` and to `BreitWigner(location, 2 * scale)`.

Domain/Support: `reals`/`reals`.

Parameters:

- `location = elementof(reals)`: location parameter x_0.
- `scale = elementof(posreals)`: scale parameter γ.

Density w.r.t. `Lebesgue(reals)`:

$$\frac{1}{\pi\gamma\left(1+\left(\frac{x-x_0}{\gamma}\right)^2\right)}$$

`StudentT(nu)` — Student's t-distribution (standard form, zero mean, unit scale).

Domain/Support: `reals`/`reals`.

Parameters:

- `nu = elementof(posreals)`: degrees of freedom ν.

Density w.r.t. `Lebesgue(reals)`:

$$\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\,\Gamma\left(\frac{\nu}{2}\right)}\left(1+\frac{x^2}{\nu}\right)^{-(\nu+1)/2}$$

The location-scale form is obtained via `pushfwd(fn(mu + sigma * _), StudentT(nu))`.

`StudentT(1)` is equivalent to `Cauchy(0, 1)`, and `StudentT(inf)` is equivalent to `Normal(0, 1)`.

`Logistic(mu, s)` — The logistic distribution.

Domain/Support: `reals`/`reals`.

Parameters:

- `mu = elementof(reals)`: location μ.
- `s = elementof(posreals)`: scale s.

Density w.r.t. `Lebesgue(reals)`:

$$\frac{e^{-(x-\mu)/s}}{s\left(1+e^{-(x-\mu)/s}\right)^2}$$

`LogNormal(mu, sigma)` — The log-normal distribution. If $X \sim \text{LogNormal}(\mu, \sigma)$, then $\log(X) \sim \text{Normal}(\mu, \sigma)$.

Domain/Support: `reals`/`posreals`.

Parameters:

- `mu = elementof(reals)`: log-space mean μ.
- `sigma = elementof(posreals)`: log-space standard deviation σ.

Density w.r.t. `Lebesgue(reals)`:

$$\frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right) \quad \text{for} \quad x > 0$$

`LogNormal(mu, sigma)` is equivalent to `pushfwd(exp, Normal(mu, sigma))`.

`Exponential(rate)` — The exponential distribution.

Domain/Support: `reals`/`nonnegreals`.

Parameters:

- `rate = elementof(posreals)`: the decay rate λ.

Density w.r.t. `Lebesgue(reals)`:

$$\lambda\, e^{-\lambda x} \quad \text{for} \quad x \geq 0$$

`Gamma(shape, rate)` — The gamma distribution.

Domain/Support: `reals`/`posreals`.

Parameters:

- `shape = elementof(posreals)`: shape parameter α.
- `rate = elementof(posreals)`: rate parameter β (inverse of scale).

Density w.r.t. `Lebesgue(reals)`:

$$\frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x} \quad \text{for} \quad x > 0$$

The chi-squared distribution with k degrees of freedom is `Gamma(shape = k/2, rate = 0.5)`.

`Weibull(shape, scale)` — The Weibull distribution. Generalizes the exponential distribution; `Weibull(1, 1/rate)` is equivalent to `Exponential(rate)`.

Domain/Support: `reals`/`nonnegreals`.

Parameters:

- `shape = elementof(posreals)`: shape parameter k.
- `scale = elementof(posreals)`: scale parameter λ.

Density w.r.t. `Lebesgue(reals)`:

$$\frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k} \quad \text{for} \quad x \geq 0$$

`InverseGamma(shape, scale)` — The inverse-gamma distribution. If $X \sim \text{Gamma}(\alpha, \beta)$, then $1/X \sim \text{InverseGamma}(\alpha, 1/\beta)$. Conjugate prior for the variance of a normal distribution.

Domain/Support: `reals`/`posreals`.

Parameters:

- `shape = elementof(posreals)`: shape parameter α.
- `scale = elementof(posreals)`: scale parameter β.

Density w.r.t. `Lebesgue(reals)`:

$$\frac{\beta^\alpha}{\Gamma(\alpha)} x^{-\alpha-1} e^{-\beta/x} \quad \text{for} \quad x > 0$$

`InverseGamma(shape, scale)` is equivalent to `pushfwd(fn(1/_), Gamma(shape, scale))`.

`Beta(alpha, beta)` — The beta distribution.

Domain/Support: `reals`/`unitinterval`.

Parameters:

- `alpha = elementof(posreals)`: shape parameter α.
- `beta = elementof(posreals)`: shape parameter β.

Density w.r.t. `Lebesgue(reals)`:

$$\frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)} \quad \text{for} \quad x \in (0, 1)$$

8.2 Standard discrete distributions

Distribution	Parameters	Domain	Support
`Bernoulli`	`p`	`integers`	`booleans`
`Categorical`	`p`	`nonnegintegers`	`interval(0, n-1)`
`Binomial`	`n`, `p`	`integers`	`interval(0, n)`
`Poisson`	`rate`	`integers`	`nonnegintegers`
`ContinuedPoisson`	`rate`	`reals`	`nonnegreals`

`Bernoulli(p)` — The Bernoulli distribution.

Domain/Support: `integers`/`booleans`.

Parameters:

- `p = elementof(unitinterval)`: success probability.

Density w.r.t. `Counting(integers)`:

$$p^k(1-p)^{1-k} \quad \text{for} \quad k \in \{0, 1\}$$

`Categorical(p)` — The categorical distribution over n categories. Generalizes the Bernoulli distribution; `Categorical([1-p, p])` is equivalent to `Bernoulli(p)`.

Domain/Support: `integers`/`interval(1, n)`.

Parameters:

- `p = elementof(stdsimplex(n))`: probability vector. Use `l1unit(weights)` or `softmax(logweights)` to construct from unnormalized weights.

Density w.r.t. `Counting(integers)`:

$$p_k \quad \text{for} \quad k \in \{1, ..., n\}$$

Categories are numbered starting from 1, consistent with FlatPPL's 1-based indexing convention.

`Binomial(n, p)` — The binomial distribution.

Domain/Support: `integers`/`interval(0, n)`.

Parameters:

- `n = elementof(posintegers)`: number of trials.
- `p = elementof(unitinterval)`: success probability.

Density w.r.t. `Counting(integers)`:

$$\binom{n}{k} p^k (1-p)^{n-k} \quad \text{for} \quad k \in \{0, ..., n\}$$

`Poisson(rate)` — The Poisson distribution.

Domain/Support: `integers`/`nonnegintegers`.

Parameters:

- `rate = elementof(nonnegreals)`: expected count λ.

Density w.r.t. `Counting(integers)`:

$$\frac{\lambda^k e^{-\lambda}}{k!} \quad \text{for} \quad k \in \mathbb{N}_0$$

At $\lambda = 0$, the distribution is the Dirac measure at $k = 0$. The parameter is called `rate` since `lambda` is a Python keyword.

For natively binned models, `broadcast(Poisson, expected_counts)` produces an array-valued observation kernel of independent Poisson counts.

`ContinuedPoisson(rate)` — Continuous extension of `Poisson` to the reals. `ContinuedPoisson` is not normalized, and so not a probability measure. At non-negative integer values, its density w.r.t. the Lebesgue measure is same as the density of `Poisson` w.r.t. the counting measure, with a continuous extension in between (by replacing the Poisson factorial with the gamma function). `ContinuedPoisson` is popular in particle physics to obtain a well-defined "Poisson-like" log-density evaluation on non-integer data such as Asimov datasets. `draw(ContinuedPoisson(rate))` is not a well-defined operation in FlatPPL.

Domain/Support: `reals`/`nonnegreals`.

Parameters:

- `rate = elementof(nonnegreals)`: expected count λ.

Density w.r.t. `Lebesgue(reals)`:

$$\frac{\lambda^x e^{-\lambda}}{\Gamma(x+1)} \quad \text{for} \quad x \geq 0$$

8.3 Multivariate distributions

Distribution	Parameters	Domain	Support
`MvNormal`	`mu`, `cov`	`cartpow(reals, n)`	`cartpow(reals, n)`
`Wishart`	`nu`, `scale`	matrices	pos. definite matrices
`InverseWishart`	`nu`, `scale`	matrices	pos. definite matrices
`LKJ`	`n`, `eta`	matrices	correlation matrices
`LKJCholesky`	`n`, `eta`	matrices	lower-triangular, pos. diagonal
`Dirichlet`	`alpha`	`cartpow(reals, n)`	`stdsimplex(n)`
`Multinomial`	`n`, `p`	`cartpow(integers, k)`	(see below)

`MvNormal(mu, cov)` — The multivariate normal distribution.

Domain/Support: `cartpow(reals, n)`/`cartpow(reals, n)`.

Parameters:

- `mu`: mean vector (array of reals, length n).
- `cov`: covariance matrix ($n \times n$, positive definite).

Density w.r.t. `iid(Lebesgue(reals), n)`:

$$\frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$

`MvNormal(mu, cov)` is equivalent to `pushfwd(fn(mu + lower_cholesky(cov) * _), iid(Normal(0, 1), n))`.

`Wishart(nu, scale)` — The Wishart distribution, a distribution over $n \times n$ positive-definite matrices.

Domain/Support: $n \times n$ matrices / positive-definite $n \times n$ matrices.

Parameters:

- `nu = elementof(posreals)`: degrees of freedom ($\nu \geq n$).
- `scale`: scale matrix ($n \times n$, positive definite).

Density w.r.t. Lebesgue on the space of $n \times n$ symmetric matrices:

$$\frac{|\mathbf{X}|^{(\nu-n-1)/2} \exp\left(-\frac{1}{2}\operatorname{tr}\left(\mathbf{V}^{-1}\mathbf{X}\right)\right)}{2^{\nu n/2}|\mathbf{V}|^{\nu/2}\Gamma_n(\nu/2)}$$

where $\mathbf{V}$ is the scale matrix and Γ_n is the multivariate gamma function.

`Wishart` is the conjugate prior for the precision matrix (inverse covariance) of `MvNormal`.

`InverseWishart(nu, scale)` — The inverse Wishart distribution, a distribution over $n \times n$ positive-definite matrices.

Domain/Support: $n \times n$ matrices / positive-definite $n \times n$ matrices.

Parameters:

- `nu = elementof(posreals)`: degrees of freedom ($\nu \geq n$).
- `scale`: scale matrix ($n \times n$, positive definite).

Density w.r.t. Lebesgue on the space of $n \times n$ symmetric matrices:

$$\frac{|\boldsymbol{\Psi}|^{\nu/2}|\mathbf{X}|^{-(\nu+n+1)/2} \exp\left(-\frac{1}{2}\operatorname{tr}\left(\boldsymbol{\Psi}\mathbf{X}^{-1}\right)\right)}{2^{\nu n/2}\Gamma_n(\nu/2)}$$

where $\boldsymbol{\Psi}$ is the scale matrix and Γ_n is the multivariate gamma function.

`InverseWishart` is the conjugate prior for the covariance matrix of `MvNormal`. `InverseWishart(nu, scale)` is equivalent to `pushfwd(inv, Wishart(nu, inv(scale)))`.

`LKJ(n, eta)` — The LKJ distribution (Lewandowski, Kurowicka, Joe) over $n \times n$ correlation matrices. Uniform over correlation matrices when $\eta = 1$; concentrates toward the identity as η increases.

Domain/Support: $n \times n$ matrices / $n \times n$ correlation matrices (symmetric, positive definite, unit diagonal).

Parameters:

- `n = elementof(posintegers)`: matrix dimension.
- `eta = elementof(posreals)`: shape parameter.

`LKJ(n, eta)` is equivalent to `pushfwd(row_gram, LKJCholesky(n, eta))`.

`LKJCholesky(n, eta)` — The lower-triangular Cholesky-factor form of the LKJ distribution. Variates are $n \times n$ lower-triangular matrices with positive diagonal entries.

Domain/Support: $n \times n$ matrices / lower-triangular $n \times n$ matrices with positive diagonal and unit-norm rows.

Parameters:

- `n = elementof(posintegers)`: matrix dimension.
- `eta = elementof(posreals)`: shape parameter.

`Dirichlet(alpha)` — The Dirichlet distribution, the multivariate generalization of the Beta distribution.

Domain/Support: `cartpow(reals, n)`/`stdsimplex(n)`.

Parameters:

- `alpha`: concentration parameters (array of positive reals, length `n`).

Density w.r.t. `Lebesgue(stdsimplex(n))`:

$$\frac{\Gamma(\sum_i \alpha_i)}{\prod_i \Gamma(\alpha_i)} \prod_i x_i^{\alpha_i - 1}$$

`Multinomial(n, p)` — The multinomial distribution, the multivariate generalization of the Binomial distribution. `Multinomial(n, [1-p, p])` is equivalent to a reparameterized `Binomial(n, p)`.

Domain/Support: `cartpow(integers, k)` / $\left\{x \in \mathbb{N}_0^k : \sum_i x_i = n\right\}$.

Parameters:

- `n = elementof(posintegers)`: number of trials.
- `p = elementof(stdsimplex(k))`: probability vector.

Density w.r.t. `iid(Counting(integers), k)`:

$$\frac{n!}{\prod_i x_i!} \prod_i p_i^{x_i} \quad \text{for} \quad x_i \geq 0, \sum_i x_i = n$$

8.4 Composite distributions

Distribution	Parameters	Domain	Support
`PoissonProcess`	`intensity`	arrays/tables	arrays/tables
`BinnedPoissonProcess`	`bins`, `intensity`	integer arrays	integer arrays

`PoissonProcess(intensity)` — The (inhomogeneous) Poisson point process, parameterized by an intensity measure. Variates are arrays (scalar points) or tables (record-valued points). The order of entries in the resulting array or table carries no semantic meaning (permutation-invariant).

Domain/Support: arrays/tables.

Parameters:

- `intensity`: finite-mass measure or kernel over scalar or record-valued points.

Given a normalized distribution `shape` and an expected count `n`, the intensity is constructed via `weighted(n, shape)`. Conversely, any intensity decomposes as `totalmass(intensity)` (expected count) and `normalize(intensity)` (shape distribution).

For binned models, see `BinnedPoissonProcess`.

Note. In particle physics, a likelihood based on a Poisson process is often called an extended likelihood.

`BinnedPoissonProcess(bins, intensity)` — Binned Poisson process: the pushforward of a `PoissonProcess` through `bincounts`. Variates are integer count arrays (one count per bin).

Domain/Support: integer arrays / integer arrays.

Parameters:

- `bins`: bin edges (vector) or record of bin edge vectors (multi-dimensional binning). Same format as for `bincounts`.
- `intensity`: finite-mass measure or kernel over the underlying event space (scalar or record-valued), not the binned count space. See `PoissonProcess`.

`BinnedPoissonProcess(bins, intensity)` is equivalent to `pushfwd(fn(bincounts(bins, _)), PoissonProcess(intensity))`.

For natively binned models where expected counts per bin are computed directly, `broadcast(Poisson, expected_counts)` is the more natural form (see `Poisson`).

8.5 Particle physics distributions

Distribution	Parameters	Domain	Support
`CrystalBall`	`m0`, `sigma`, `alpha`, `n`	`reals`	`reals`
`DoubleSidedCrystalBall`	`m0`, `sigmaL`, `sigmaR`, `alphaL`, `nL`, `alphaR`, `nR`	`reals`	`reals`
`Argus`	`resonance`, `slope`, `power`	`reals`	`interval(0, resonance)`

Distribution	Parameters	Domain	Support
`BreitWigner`	`mean`, `width`	`reals`	`reals`
`RelativisticBreitWigner`	`mean`, `width`	`reals`	`posreals`
`Voigtian`	`mean`, `width`, `sigma`	`reals`	`reals`
`BifurcatedGaussian`	`mean`, `sigmaL`, `sigmaR`	`reals`	`reals`

`CrystalBall(m0, sigma, alpha, n)` — The Crystal Ball distribution: Gaussian core with a power-law tail on one side.

Domain/Support: `reals`/`reals`.

Parameters:

- `m0 = elementof(reals)`: peak position.
- `sigma = elementof(posreals)`: width.
- `alpha = elementof(posreals)`: transition point (in units of σ).
- `n = elementof(posreals)`: power-law exponent.

`DoubleSidedCrystalBall(m0, sigmaL, sigmaR, alphaL, nL, alphaR, nR)` — The double-sided Crystal Ball distribution: Gaussian core with independent power-law tails on both sides.

Domain/Support: `reals`/`reals`.

Parameters:

- `m0 = elementof(reals)`: peak position.
- `sigmaL = elementof(posreals)`, `sigmaR = elementof(posreals)`: left/right widths.
- `alphaL = elementof(posreals)`, `alphaR = elementof(posreals)`: left/right transition points.
- `nL = elementof(posreals)`, `nR = elementof(posreals)`: left/right power-law exponents.

`Argus(resonance, slope, power)` — The ARGUS distribution.

Domain/Support: `reals`/`interval(0, resonance)`.

Parameters:

- `resonance = elementof(posreals)`: kinematic endpoint.
- `slope = elementof(reals)`: slope parameter.
- `power = elementof(posreals)`: power parameter (typically 0.5).

`BreitWigner(mean, width)` — The non-relativistic Breit-Wigner (Cauchy/Lorentzian) distribution, parameterized by resonance position and full width. Equivalent to `Cauchy(mean, width / 2)`. The non-relativistic and relativistic Breit-Wigners are distinct distributions and have separate constructors.

Domain/Support: `reals`/`reals`.

Parameters:

- `mean = elementof(reals)`: resonance position m.
- `width = elementof(posreals)`: full width at half maximum Γ.

Density w.r.t. `Lebesgue(reals)`:

$$\frac{1}{\pi}\frac{\Gamma/2}{(x-m)^2+(\Gamma/2)^2}$$

`RelativisticBreitWigner(mean, width)` — The relativistic Breit-Wigner distribution.

Domain/Support: `reals`/`posreals`.

Parameters:

- `mean = elementof(posreals)`: resonance mass m.
- `width = elementof(posreals)`: full width Γ.

`Voigtian(mean, width, sigma)` — The Voigt profile: convolution of a Breit-Wigner and a Gaussian.

Domain/Support: `reals`/`reals`.

Parameters:

- `mean = elementof(reals)`: resonance position.
- `width = elementof(posreals)`: Breit-Wigner full width Γ.
- `sigma = elementof(posreals)`: Gaussian resolution.

`BifurcatedGaussian(mean, sigmaL, sigmaR)` — Split normal distribution: Gaussian with different widths on left and right sides.

Domain/Support: `reals`/`reals`.

Parameters:

- `mean = elementof(reals)`: peak position.
- `sigmaL = elementof(posreals)`: left-side width.
- `sigmaR = elementof(posreals)`: right-side width.

---

9 Worked examples

9.1 High Energy Physics (HEP)

This example walks through a realistic HEP model step by step.

Signal and background model. We begin with a systematic uncertainty on the signal efficiency, modeled as a unit-normal nuisance parameter:

```
raw_eff_syst = draw(Normal(mu = 0.0, sigma = 1.0))
efficiency = 0.9 + 0.05 * raw_eff_syst
```

Signal and background shapes are defined as step-function densities, normalized over the analysis region:

```
sig_shape = fn(stepwise(bin_edges = bin_edges, bin_values = signal_bins, x =
_))
bkg_shape = fn(stepwise(bin_edges = bin_edges, bin_values = bkg_bins, x = _))
signal_template = normalize(weighted(sig_shape, Lebesgue(support =
interval(lo, hi))))
bkg_template = normalize(weighted(bkg_shape, Lebesgue(support = interval(lo,
hi))))
```

Observation model. The rate measure superposes signal (scaled by signal strength `mu_sig` and efficiency) with background. The module input `mu_sig = elementof(reals)` plays the role of the model's parameter of interest. Events are drawn from a Poisson point process:

```
rate = superpose(
    weighted(mu_sig * efficiency, signal_template),
    bkg_template
)
events = draw(PoissonProcess(intensity = rate))
```

Data and likelihood. We define observed data and construct the likelihood. Since the event space is scalar, the `PoissonProcess` produces an array variate and the observed data is a plain array. The observation model uses `lawof` with a boundary input to keep `raw_eff_syst` as a kernel parameter (rather than marginalizing it out). A separate constraint term represents the auxiliary measurement that pins the nuisance parameter. The combined likelihood `L` is a likelihood object on the parameter space {`mu_sig`, `raw_eff_syst`}:

```
# Observation likelihood: boundary input keeps raw_eff_syst as a parameter
L_obs = likelihoodof(
    lawof(events, raw_eff_syst = raw_eff_syst),
    [3.1, 5.7, 2.4, 8.9, 4.2])

# Constraint: auxiliary measurement model for the nuisance parameter
aux_eff = draw(Normal(mu = raw_eff_syst, sigma = 1.0))
L_constr = likelihoodof(lawof(aux_eff, raw_eff_syst = raw_eff_syst), 0.0)

# Combined likelihood
L = joint_likelihood(L_obs, L_constr)
```

The constraint likelihood $L_{\text{constr}}(\alpha) = \varphi(0; \alpha, 1)$ is a genuine function of `raw_eff_syst` — the auxiliary observation model `Normal(mu = raw_eff_syst, sigma = 1.0)` is a kernel parameterized

by the nuisance parameter, and `likelihoodof` evaluates its density at the auxiliary datum 0.0. (By Normal symmetry, $\varphi(0;\alpha,1) = \varphi(\alpha;0,1)$, so numerically this gives the standard Gaussian penalty. But the semantic structure matters: the constraint is a likelihood term, not a prior.)

A frequentist engine can maximize `L` or compute profile likelihood ratios. A range-restricted likelihood for a sideband fit is also straightforward:

```
sideband = interval(0.0, 3.0)
sideband_data = filter(fn(_ in sideband), [3.1, 5.7, 2.4, 8.9, 4.2])
sideband_model = normalize(truncate(lawof(events, raw_eff_syst =
raw_eff_syst), sideband))
L_obs_sideband = likelihoodof(sideband_model, sideband_data)
L_sideband = joint_likelihood(L_obs_sideband, L_constr)
```

Bayesian analysis (optional). To construct a posterior, define priors and reweight:

```
mu_sig_prior = draw(Uniform(support = interval(0, 20)))
raw_eff_syst_prior = draw(Normal(mu = 0, sigma = 1))
prior = lawof(record(mu_sig = mu_sig_prior, raw_eff_syst =
raw_eff_syst_prior))
posterior = bayesupdate(L, prior)
# posterior is unnormalized; wrap in normalize(...) if needed
```

Additional patterns. The following snippets illustrate further language features in the context of the same analysis style — variate naming, variable transformations, broadcast, truncation, density-defined distributions, module loading, and hypothesis testing:

```
# Variate naming with pushfwd
mvmodel = pushfwd(fn(relabel(_, ["a", "b", "c"])), MvNormal(mu = some_mean,
cov = some_cov))
L_mv = likelihoodof(mvmodel, record(a = 1.1, b = 2.1, c = 3.1))

# Expanded form (when intermediate variates are needed)
a, b, c = draw(MvNormal(mu = some_mean, cov = some_cov))
mvmodel_expanded = lawof(record(a = a, b = b, c = c))

# Pushforward for variable transformation
log_normal = pushfwd(functionof(exp(x), x = x), Normal(mu = 0, sigma = 1))

# Deterministic function and broadcast
transformed = 2 * a + 1
f = functionof(transformed, a = a)
A = [1.0, 2.0, 3.0, 4.0]
result = broadcast(f, a = A)           # [3.0, 5.0, 7.0, 9.0]
result = broadcast(f, A)              # same, positional (f has declared
order)
```

```
# Stochastic broadcast
noisy = draw(Normal(mu = a, sigma = 0.1))
K = lawof(noisy)
noisy_array = draw(broadcast(K, a = A))  # independent Normal draws at each
element

# Truncated distribution (model physics)
positive_sigma = draw(normalize(truncate(Normal(mu = 1.0, sigma = 0.5),
interval(0, inf))))

# Density-defined distribution (Bernstein polynomial)
bern = fn(bernstein(coefficients = [c0, c1, c2, c3], x = _))
smooth_bkg = normalize(weighted(bern, Lebesgue(support = interval(lo, hi))))

# Module loading and composition
sig = load_module("signal_channel.flatppl")
bkg = load_module("background_channel.flatppl")
L_combined = joint_likelihood(
    likelihoodof(sig.model, sig.data),
    likelihoodof(bkg.model, bkg.data)
)

# Hypothesis testing (two models, same data, explicit IID)
model_H0 = iid(Normal(mu = 91.2, sigma = 2.5), 4)
model_H1 = iid(Normal(mu = 125.0, sigma = 3.0), 4)
mass_data = [90.1, 91.8, 124.5, 125.2]
L_H0 = likelihoodof(model_H0, mass_data)
L_H1 = likelihoodof(model_H1, mass_data)
```

---

10 Intermediate Representation

This section defines **FlatPIR**, the intermediate representation of FlatPPL. FlatPPL engines may ingest either FlatPPL or FlatPIR, depending on their design.

Note: The design of FlatPIR is preliminary and subject to change. It is not part of FlatPPL semantic versioning yet.

Like FlatPPL, FlatPIR comes with a canonical syntax. The canonical FlatPIR syntax uses standard S-expressions (compatible with Lisp/Scheme readers).

FlatPIR is designed to support term-rewriting, with two main use cases:

- Restricting FlatPPL/FlatPIR code to a specific subset that maps directly to a target probabilistic language (see Profiles and interoperability).
- Optimizing FlatPPL/FlatPIR code before handing it off to host-language implementations (which then can do further optimization within their own language stack).

Term-rewriting can require both value type and binding phase (see Phases) information, so FlatPIR includes `(%meta (%type ...) (%phase ...))` metadata annotations on every binding.

The semantics of FlatPIR are identical to the semantics of FlatPPL, with the addition of binding metadata. They are independent of the canonical S-expression representation. Additional representations (e.g. binary) are expected for some use cases but are not yet specified.

FlatPPL maps directly to FlatPIR and FlatPIR maps back directly to FlatPPL; type and phase metadata are dropped in the process.

10.1 Naming convention

FlatPIR structural keywords are prefixed with `%` (e.g. `%module`, `%bind`, `%ref`, `%type`). FlatPPL built-in names (`Normal`, `add`, `record`, `vector`, `real`, `integer`, ...) and user-defined names appear bare. The `%` prefix is invalid in FlatPPL syntax (not Python/Julia AST compatible), so FlatPIR structural keywords cannot collide with FlatPPL built-in and binding names.

10.2 Module structure

Each surface FlatPPL module (file or embedded code block) maps to one FlatPIR `(%module ...)`; modules are not flattened in FlatPIR, though tooling may flatten them internally (e.g. for cross-module optimization before code evaluation). FlatPIR files use the file extension `.flatpir`. A FlatPIR file contains exactly one `(%module ...)` form with these elements:

- `(%meta ...)` — module-level metadata, including `flatppl_compat` version.
- `(%load <module> (%path "..."))` — zero or more declarations of loaded dependencies, optionally followed by a `(%bindings ...)` sub-form supplying substitution values for the loaded module's free inputs.
- `(%exports <name1> <name2> ...)` — the module's public interface. Bindings listed here are the root set for rewriting passes; unlisted bindings may be elided during term-rewriting.
- `(%bind <name> <expression> (%meta (%type <t>) (%phase <p>)))` — pairs a name with an expression, a type annotation, and a phase annotation. Before inference both annotations are `%deferred` (see below).

Top-level declarations may appear in any order: bindings are resolved by reference, not by textual position.

A parameterized load looks like:

```
(%load helpers (%path "helpers.flatppl")
       (%bindings
         (%assign center (%ref %global a))))
```

Each substitution takes the form `(%assign <input-name> <expression>)`. The expression is resolved in the loading module's namespace.

10.3 Type and phase annotations

Every binding in FlatPIR carries `(%meta (%type ...) (%phase ...))` metadata. Before inference, both slots hold `%deferred`; inference rewrites them in place with concrete types and binding phases. Phase is computed by ancestor analysis (cheaper than type inference): `%fixed`, `%parameterized`, or `%stochastic` (see Phases).

FlatPPL is designed such that type inference on a well-formed module can always succeed. If inference fails — for example, an unresolvable reference or a type error in an expression — the module is ill-formed and the engine should report a static error. As a diagnostic aid, the engine may also rewrite the affected `(%type %deferred)` slot to `(%type (%failed "reason"))`, so that downstream tooling and users can see the cause and location of the failure inline.

The "type" terminology refers to the **structural category** of a value — scalar, array, record, table, measure, kernel, likelihood, function — not to a type system in the traditional programming-language sense.

Sets and types are distinct. Set membership information attached via `elementof` (e.g. `(elementof posreals)`) is preserved structurally in the expression itself, not encoded into the type annotation. The type annotation records structural category (e.g. `(%scalar real)`); the `elementof` expression records set membership (e.g. `posreals` as a subset of `reals`).

10.3.1 Type categories

- `%deferred` — pipeline-state placeholder for "not yet resolved at this stage." Appears as a binding's top-level type before inference has run.
- `(%failed "<reason>")` — diagnostic marker written into a binding's `(%type ...)` slot when inference attempted to resolve it but could not. The reason string is for human and tooling consumption. A module containing any `%failed` marker is ill-formed.
- `%any` — used where no concrete-type constraint is applicable, e.g. for the input of `fn(sum(_))`. Counterpart of the value-level set `anything`.
- `(%scalar real)`, `(%scalar integer)`, `(%scalar boolean)`, `(%scalar complex)` — the four scalar value types.
- `(%array <rank> <shape> <element-type>)` — arrays. `<rank>` is a positive integer literal (not `%dynamic`). Each entry in `<shape>` is a positive integer dimension size, or the placeholder `%dynamic` for a dimension whose size is determined at load or runtime rather than statically (e.g. `(%array 2 (%dynamic 3) (%scalar real))` is a 2D real array with three columns and a dynamic row count).
- `(%record (<field> <type>) ...)` — records with named fields.
- `(%table (%columns (<name> <type>) ...) (%nrows <N>))` — tables with named columns and row count. `<N>` is a positive integer or `%dynamic`; tables loaded via `load_data` are a common source of dynamic row counts.
- `(%tuple <type1> <type2> ...)` — tuples with at least two elements.
- `(%measure (%domain <type>))` — closed measures. `<type>` is the type of values that sampling generates and on which density evaluation is defined.

- `(%kernel (%inputs (<name> <type>) ...) (%domain <type>))` — transition kernels. `(%domain <type>)` corresponds to the domain of the closed measures generated by the kernel.
- `(%function (%inputs (<name> <type>) ...) (%result <type>))` — functions.
- `(%likelihood (%inputs (<name> <type>) ...) (%obstype <type>))` — likelihood objects.

10.4 Expressions

Expressions in FlatPIR come in structurally distinct shapes for built-in operations, references, and calls to user-defined callables. Rewriting rules can pattern-match on expression category without name-based dispatch.

Built-in operations are bare-headed forms with the operation name as the head symbol:

```
(add x y)
(Normal (%kwarg mu (real 0.0)) (%kwarg sigma (real 1.0)))
(draw (Normal ...))
(elementof reals)
(load_data (%kwarg source (string "...")) (%kwarg valueset ...))
```

Most built-in callables support both positional arguments and `%kwarg` entries, matching the surface FlatPPL form. `draw` and `elementof` are positional-only; user-defined callables reified without explicit boundary declarations are keyword-only (see calling conventions).

Some FlatPPL forms have FlatPIR shapes distinct from ordinary calls and have variadic keyword arguments which are syntactically the same or ordinary keyword arguments in surface FlatPPL, but structurally different since their order carries semantic meaning. Some of these forms also have a single leading positional argument:

- `functionof` and `lawof` take variadic kwargs that define parameters of the reified callable. FlatPIR uses `(%params ...)` for the parameter list.
- `record`, `table`, `cartprod`, `joint`, `jointchain` take variadic kwargs that label components of the output. FlatPIR uses `(%field ...)` entries (see below).
- `load_module` lowers to the top-level `(%load ...)` module element rather than a binding expression, with its kwargs as `(%assign ...)` entries inside `(%bindings ...)` (see Module structure).

Built-in constants appear as bare symbols in argument positions:

```
reals  posreals  integers  booleans  pi  inf  im
```

References to named bindings use `(%ref <namespace> <name>)`:

- `(%ref %global <name>)` — reference to global binding in the current module.
- `(%ref %local <name>)` — reference to parameter inside `functionof` or `lawof`.
- `(%ref <module> <name>)` — reference to global binding in a loaded module.

Calls to user-defined callables use `(%call <ref-head> <args>...)`:

```
(%call (%ref %global helper_fn) x y)
(%call (%ref helpers obs_kernel) row)
```

User bindings always use `(%ref ...)` while built-ins use bare symbols. This is unambiguous because FlatPPL does not allow user bindings to shadow built-in names.

A rewriter pattern on `(%call ?head ?args...)` fires only on user-defined calls; a pattern on `(add ?x ?y)` fires only on the built-in. There is no overlap.

Positional and keyword call forms. Built-in operations and user-defined calls may use positional arguments or `%kwarg` entries, matching the surface FlatPPL form. Both are valid FlatPIR with identical semantics for a given callable. `%kwarg` entries are unordered: `(Normal (%kwarg sigma (real 1.0)) (%kwarg mu (real 0.0)))` is the same call as `(Normal (%kwarg mu (real 0.0)) (%kwarg sigma (real 1.0)))`.

Structural named entries use two dedicated heads distinct from `%kwarg`:

- `(%field <name> <value>)` — named entries in data constructors (e.g., `record`, `cartprod`, `joint`, `table`). Order is part of the structure.
- `(%assign <name> <value>)` — substitutions and interface bindings (e.g., `%load` bindings). Unordered (matched by name).

Literal values. Scalar literals use FlatPPL scalar restriction and constructor function names as heads, followed by bare atom values:

```
(integer 3)
(real 1.0)
(complex 0.5 2.0)
(string "inputs.csv")
(boolean true)
(vector (real 1.0) (real 2.0) (real 3.0))
(record (%field mu (real 0.0)) (%field sigma (real 1.0)))
```

`vector` and `record` literals follow the same pattern.

The `vector` form is `(vector <expr>...)`. Each element is a full expression (tagged literal, reference, or built-in call):

```
(vector (real 1.0) (%ref %global a) (real 2.0))       ; mixes literal and
reference
(vector (%ref %global a) (%ref %global b))            ; pre-inference;
elements are expressions
```

Vectors of vectors:

```
(vector
  (vector (real 1.0) (real 2.0) (real 3.0))
  (vector (real 4.0) (real 5.0)))
```

Complex elements:

```
(vector (complex 0.5 2.0) (complex 1.0 0.0))
```

The `tuple` form is `(tuple <expr>...)` with at least two elements. Unlike `vector`, component types may differ and may include non-value objects (functions, measures, kernels, likelihoods):

```
(tuple (%ref %global forward_kernel) (%ref %global prior))
```

Tuple decomposition on the surface (`a, b = expr`) lowers to successive `(get ...)` projections with integer indices.

Function parameter lists. `functionof` and `lawof` introduce explicit parameter lists via `(%params ...)`:

```
(functionof (%params (center spread _x_))
  (Normal (%kwarg mu (add (%ref %local center) (%ref %local _x_)))
          (%kwarg sigma (%ref %local spread))))
```

Inside the body, parameter references use `(%ref %local <name>)`. Parameter names preserve the surface trailing-underscore placeholder convention (e.g. _x_), keeping the round-trip to surface FlatPPL trivial.

Normalization. Bare FlatPIR preserves the surface calling convention for round-trip fidelity. Optional normalization passes can convert keyword arguments to positional where the argument order is known (built-ins, explicitly-ordered user callables) and sort remaining keyword arguments into canonical order. Normalized FlatPIR is easier for term-rewriting systems to pattern-match and deduplicate.

10.5 Cross-module type inference

Each module is annotated independently: types are computed from its own perspective (using `%global` for module-level references). When module B loads module A, B's inference proceeds as follows:

1. For each `(%load <module> (%path "..."))`, locate A's `.flatpir` file.
2. If A is not yet annotated, run inference on it first (with cycle detection).
3. Read A's exported bindings and their type annotations.
4. Translate A's `%global` references: each `(%ref %global X)` becomes `(%ref <module> X)` unless the load supplies a substitution for `X`, in which case the substitution expression replaces the reference entirely.

5. Use A's translated annotations when resolving cross-module references in B. When an exported type contains `%any` (e.g. a generic function), B's inference flows B's concrete argument types through A's function body to determine the concrete result type at each call site.

A's annotation file is read-only from B's perspective; the same annotated file serves every caller. Type annotations are sufficient for term rewriting within a module; cross-module type inference may additionally traverse exported function bodies when signatures contain `%any`.

10.6 Example

A two-module example showing lowering and annotation.

10.6.1 Surface FlatPPL

`helpers.flatppl`:

```
center = elementof(reals)
spread = elementof(posreals)

obs_kernel = functionof(
    Normal(mu = center + _x_, sigma = spread),
    center = center, spread = spread, x = _x_)

shifted_value = center + 1.0
```

`model.flatppl`:

```
a = elementof(reals)
helpers = load_module("helpers.flatppl", center = a)

b = draw(Normal(mu = 0.0, sigma = 2.0))
_combined = a + b

input_data = load_data(
  source = "inputs.csv",
  valueset = cartprod(x = reals)
)

L = likelihoodof(helpers.obs_kernel, input_data)
```

10.6.2 Bare FlatPIR

`helpers.flatpir`:

```
(%module
  (%meta (flatppl_compat "0.6"))
  (%exports center spread obs_kernel shifted_value)

  (%bind center
```

```
    (elementof reals)
    (%meta (%type %deferred) (%phase %deferred)))

  (%bind spread
    (elementof posreals)
    (%meta (%type %deferred) (%phase %deferred)))

  (%bind obs_kernel
    (functionof (%params (center spread _x_))
      (Normal
        (%kwarg mu (add (%ref %local center) (%ref %local _x_)))
        (%kwarg sigma (%ref %local spread))))
    (%meta (%type %deferred) (%phase %deferred)))

  (%bind shifted_value
    (add (%ref %global center) (real 1.0))
    (%meta (%type %deferred) (%phase %deferred))))
```

`model.flatpir`:

```
(%module
  (%meta (flatppl_compat "0.6"))
  (%exports a b input_data L)

  (%load helpers (%path "helpers.flatppl")
         (%bindings (%assign center (%ref %global a))))

  (%bind a
    (elementof reals)
    (%meta (%type %deferred) (%phase %deferred)))

  (%bind b
    (draw (Normal (%kwarg mu (real 0.0)) (%kwarg sigma (real 2.0))))
    (%meta (%type %deferred) (%phase %deferred)))

  (%bind _combined
    (add (%ref %global a) (%ref %global b))
    (%meta (%type %deferred) (%phase %deferred)))

  (%bind input_data
    (load_data
      (%kwarg source (string "inputs.csv"))
      (%kwarg valueset (cartprod (%field x reals))))
    (%meta (%type %deferred) (%phase %deferred)))

  (%bind L
```

```
    (likelihoodof (%ref helpers obs_kernel) (%ref %global input_data))
    (%meta (%type %deferred) (%phase %deferred))))
```

10.6.3 Annotated FlatPIR

`helpers.flatpir` after type inference:

```
(%module
  (%meta (flatppl_compat "0.6"))
  (%exports center spread obs_kernel shifted_value)

  (%bind center
    (elementof reals)
    (%meta (%type (%scalar real)) (%phase %parameterized)))

  (%bind spread
    (elementof posreals)
    (%meta (%type (%scalar real)) (%phase %parameterized)))

  (%bind obs_kernel
    (functionof (%params (center spread _x_))
      (Normal
        (%kwarg mu (add (%ref %local center) (%ref %local _x_)))
        (%kwarg sigma (%ref %local spread))))
    (%meta (%type (%kernel
                    (%inputs (center (%scalar real))
                             (spread (%scalar real))
                             (_x_ (%scalar real)))
                    (%domain (%scalar real))))
           (%phase %fixed)))

  (%bind shifted_value
    (add (%ref %global center) (real 1.0))
    (%meta (%type (%scalar real)) (%phase %parameterized))))
```

`model.flatpir` after type inference:

```
(%module
  (%meta (flatppl_compat "0.6"))
  (%exports a b input_data L)

  (%load helpers (%path "helpers.flatppl")
         (%bindings (%assign center (%ref %global a))))

  (%bind a
    (elementof reals)
    (%meta (%type (%scalar real)) (%phase %parameterized)))
```

```
(%bind b
  (draw (Normal (%kwarg mu (real 0.0)) (%kwarg sigma (real 2.0))))
  (%meta (%type (%scalar real)) (%phase %stochastic)))

(%bind _combined
  (add (%ref %global a) (%ref %global b))
  (%meta (%type (%scalar real)) (%phase %stochastic)))

(%bind input_data
  (load_data
    (%kwarg source (string "inputs.csv"))
    (%kwarg valueset (cartprod (%field x reals))))
  (%meta (%type (%table (%columns (x (%scalar real)))
                        (%nrows %dynamic)))
        (%phase %fixed)))

(%bind L
  (likelihoodof (%ref helpers obs_kernel) (%ref %global input_data))
  (%meta (%type (%likelihood
                  (%inputs (center (%scalar real))
                           (spread (%scalar real))
                           (_x_ (%scalar real)))
                  (%obstype (%table (%columns (x (%scalar real)))
                                    (%nrows %dynamic)))))
        (%phase %fixed))))
```

The likelihood `L` inherits its `%inputs` list from `obs_kernel`'s reified parameters — local names `center`, `spread`, and `_x_`, decoupled from any same-named module-level binding. A downstream tool walks the list and supplies a value for each parameter at the call site, with the matching done by name. `input_data`'s type was derived from the `valueset` argument of `load_data` without reading the file.

---

11 Profiles and interoperability

11.1 FlatPPL as an exchange platform

While full FlatPPL implementations are feasible for some languages and package ecosystems with modest effort (see appendix), a key strength of FlatPPL is its suitability as an exchange platform between probabilistic modeling systems. Rather than requiring pairwise translators between n systems — an $O(n^2)$ problem — FlatPPL enables a hub-and-spoke architecture: each system needs only one importer and one exporter, with term-rewriting within FlatPPL/FlatPIR handled by common tooling.

This approach follows established patterns in compiler and interoperability ecosystems: LLVM provides a language- and target-independent IR shared across many front ends and back ends;

MLIR generalizes this with multiple levels of IR and legalized conversion to target-specific subsets; ONNX plays a similar role for machine-learning models. FlatPPL aims to fill this role for probabilistic models.

Probabilistic modeling systems broadly fall into two paradigms: **stochastic-node systems** (Stan, Pyro, NumPyro) that build joint distributions incrementally via sampling primitives, and **measure-composition systems** (RooFit, HS³, MeasureBase.jl) that construct models via measure algebra. FlatPPL supports both paradigms natively (see variates and measures), and term-rewriting bridges between them. Profiles define the mechanically translatable fragment for each target.

11.2 Profiles

Not every target system supports all of FlatPPL. A **profile** is a named subset of FlatPPL — together with any required normalization, lowering, or raising conditions — that a given target system can accept as input. Unlike MLIR dialects, which introduce new operations and types in parallel namespaces, FlatPPL profiles are overlapping subsets of a single common language.

The output of a FlatPPL exporter (or tracing compiler) is valid FlatPPL by definition. What matters for conversion and term-rewriting is the *input profile* of the target system — the subset of FlatPPL the target can consume. We call this simply the **FlatPPL profile** of that system. Whoever writes the exporter decides which subset of a system's profile to emit.

11.3 Profile summary

The following table summarizes the FlatPPL acceptance patterns for each profile — what form the FlatPPL model must be in for the target system to consume it.

Feature	HS³/RooFit	pyhf/HistFactory	Stan
Measure algebra	required (compositional normal form)	limited (`superpose` only)	must be lowered to stochastic nodes
Stochastic nodes (`draw`)	must be raised to measure composition	accepted in constrained binned-likelihood patterns	accepted (primary form)
Explicit likelihood/constraint factors	yes	yes (primary structure)	no (joint model block)
Binned models	yes	yes (primary)	limited
Hierarchical models	yes (`jointchain`)	no	yes

11.4 HS³/RooFit profile

HS³ is a JSON-based interchange format for statistical models in HEP, with implementations in RooFit (C++, the most complete), pyhf (HistFactory subset), zfit (partial), and HS3.jl/BAT.jl (Julia, partial). RooFit is the most mature and widely deployed statistical modeling toolkit in HEP.

FlatPPL targets RooFit primarily via HS³; since HS³ has the firm goal of closing the remaining gaps to RooFit, we treat them as a single profile.

Side-by-side comparison. A simple model in FlatPPL and HS³ JSON:

FlatPPL source:

```
mu_param = elementof(reals)
sigma_param = elementof(posreals)
mass = draw(Normal(mu = mu_param, sigma = sigma_param))
nominal = preset(mu_param = 5.28, sigma_param = 0.003)
```

Corresponding HS³ JSON:

```
{
  "distributions": [
    {"name": "mass", "type": "gaussian_dist",
     "mean": "mu_param", "sigma": "sigma_param", "x": "mass_obs"}
  ],
  "parameter_points": [
    {"name": "default", "entries": [
      {"name": "mu_param", "value": 5.28},
      {"name": "sigma_param", "value": 0.003}
    ]}
  ]
}
```

Both describe the same mathematical content: two parameters with nominal values and a Gaussian distribution. Note the dual-naming in HS³ (distribution name `"mass"` vs. variate field `"x": "mass_obs"`); FlatPPL's `mass = draw(Normal(...))` eliminates this. The HS³ `parameter_points` entry corresponds to the FlatPPL `preset`.

What maps. FlatPPL models that can be raised to **compositional normal form** map to HS³/RooFit. Here, "compositional normal form" means a FlatPPL representation in which stochastic-node subgraphs have been raised to explicit measure and kernel composition (`joint`, `jointchain`, `chain`, `pushfwd`, `weighted`, etc.) — see variates and measures. This covers the large class of models whose density is tractable.

What does not map. Models that require intractable marginalization integrals (e.g., `chain` with intractable kernels) may not map to current HS³/RooFit. Context-dependent reinterpretation of parameter and observable roles (a RooFit pattern) is intentionally excluded from FlatPPL.

Round-trip expectations. Textual round-tripping is not a goal — multiple source texts can describe the same semantic graph. The round-trip guarantee is semantic: HS³ models map to the semantic graph and back. Canonical comparison should operate on the lowered normalized graph, not on raw source text.

Structured variates. In the current HS³ standard, all distribution variates are flat named tuples with globally unique entry names. FlatPPL additionally supports structured (record-valued and array-valued) variates; translators must flatten these for HS³ serialization.

Correspondence points. FlatPPL's DAG maps to RooFit's server/client dependency graph; `likelihoodof` maps to `createNLL`; `load_module` maps to workspace loading with parameter sharing.

11.4.1 HS³/RooFit distribution mapping

The following tables summarize major correspondences; they are illustrative rather than exhaustive.

FlatPPL	HS³	RooFit	Parameter notes
`Uniform`	`uniform_dist`	`RooUniform`	
`Normal`	`gaussian_dist` (also `normal_dist`)	`RooGaussian`	`mu` ⟶ `mean`
`GeneralizedNormal`	`generalized_normal_dist`	—	Names match HS³
`LogNormal`	`lognormal_dist`	`RooLognormal`	RooFit: `m0` = e^{μ}, `k` = e^{σ}
`Exponential`	`exponential_dist`	`RooExponential`	`rate` ⟶ `c` (HS³); RooFit: `c` = −`rate`
`Gamma`	—	`RooGamma`	`shape` ⟶ `gamma`, `rate` ⟶ $1/$`beta`, `mu` = 0
`Poisson`	`poisson_dist`	`RooPoisson`	`rate` ⟶ `mean` = λ
`ContinuedPoisson`	`poisson_dist` (implicit)	`RooPoisson` (`noRounding=true`)	Same parameter mapping as `Poisson`; density only, not generative
`MvNormal`	`multivariate_normal_dist`	`RooMultiVarGaussian`	`mu` ⟶ `mean` (HS³); `cov` ⟶ `covariances` (HS³)
`CrystalBall`	`crystalball_dist`	`RooCBShape`	Names match directly
`DoubleSidedCrystalBall`	`crystalball_dist` (double-sided)	`RooCrystalBall`	`sigmaL` ⟶ `sigma_L` (HS³), etc.
`Argus`	`argus_dist`	`RooArgusBG`	HS³: names match; RooFit: `resonance`

FlatPPL	HS³	RooFit	Parameter notes
			⟶ `m0`, `slope` ⟶ `c`, `power` ⟶ `p`
`BreitWigner`	—	`RooBreitWigner`	
`RelativisticBreitWigner`	`relativistic_breit_wigner_dist`		Names match HS³
`Voigtian`	—	`RooVoigtian`	
`BifurcatedGaussian`	—	`RooBifurGauss`	
`PoissonProcess`	`rate_extended_dist` / `rate_density_dist`	`RooExtendPdf` + base PDF	Decompose via `normalize`/ `totalmass`
`BinnedPoissonProcess`	`bincounts_extended_dist` / `bincounts_density_dist`	`RooExtendPdf` + binned PDF	

Density-defined distributions (`normalize(weighted(f, Lebesgue(support = S)))`) map to HS³'s `density_function_dist` / `log_density_function_dist`.

11.4.2 HS³/RooFit function mapping

FlatPPL	HS³ / RooFit / pyhf	Notes
`interp_pwlin`	HS³ `lin` / pyhf code0	Piecewise linear
`interp_pwexp`	HS³ `log` / pyhf code1	Piecewise exponential
`interp_poly2_lin`	HS³ `parabolic` / pyhf code2	Quadratic + linear extrapolation
`interp_poly6_lin`	HS³ `poly6` / pyhf code4p	6th-order + linear extrapolation
`interp_poly6_exp`	pyhf code4	6th-order + exponential extrapolation
`polynomial`	HS³ function graph	Power-series polynomial
`bernstein`	HS³ function graph	Bernstein basis polynomial
`stepwise`	HS³ function graph	Piecewise-constant
`bincounts`	Represented via HS³ axes metadata	Binning operation

11.4.3 HS³ `histfactory_dist` decomposition

HS³'s `histfactory_dist` encodes the entire HistFactory channel/sample/modifier structure as a single composite distribution. In FlatPPL, this decomposes into explicit components:

HS³ `histfactory_dist` component	FlatPPL equivalent
`axes`	Edge vectors used in `bincounts`
`samples[].data.contents`	Nominal bin-count arrays (plain values)
`samples[].modifiers[type=normfactor]`	Free parameter, multiply
`samples[].modifiers[type=normsys]`	`draw(Normal(...))` + `interp_*exp(...)` + multiply
`samples[].modifiers[type=histosys]`	`draw(Normal(...))` + `interp_*lin(...)`
`samples[].modifiers[type=shapefactor]`	Array-valued explicit input, multiply
`samples[].modifiers[type=shapesys]`	`draw(broadcast(Poisson(...)))`, multiply
`samples[].modifiers[type=staterror]`	`draw(broadcast(Normal(...)))`, multiply
`samples[].modifiers[].interpolation`	Choice of `interp_*` function
`samples[].modifiers[].constraint`	`Normal` vs `Poisson` in the `draw`
Sample stacking	Elementwise addition via `broadcast`
Per-bin Poisson observation	`broadcast(Poisson, total)`

HS³'s `mixture_dist` maps to `normalize(superpose(...))`, and `product_dist` maps to `joint(...)`.

11.5 pyhf/HistFactory profile

The pyhf/HistFactory profile is the subset of FlatPPL corresponding to HistFactory-style binned template models with explicit observation and constraint factors.

HistFactory describes binned statistical models as sums of histogram templates ("samples") within analysis regions ("channels"), with systematic uncertainties expressed as "modifiers" that transform expected bin counts. pyhf provides a pure-Python implementation with a declarative JSON specification.

FlatPPL can express the standard HistFactory channel/sample/modifier model without introducing special modifier objects. The key insight is that each modifier bundles two concerns:

- A **deterministic effect** on expected bin counts (interpolation, scaling, or per-bin multiplication).
- A **probabilistic constraint** on the controlling nuisance parameter (Gaussian, Poisson, or unconstrained).

FlatPPL separates these cleanly. The deterministic effects use interpolation functions and arithmetic; the probabilistic constraints use standard `draw` statements. The observation model wraps total expected counts in `broadcast(Poisson, expected)`.

11.5.1 Modifier mapping

pyhf / HistFactory	FlatPPL deterministic effect	FlatPPL constraint	Default interpolation	Notes
`normfactor` / `NormFactor`	`broadcast(fn(_ * _), expected, mu)`	none (free)	—	
`lumi`	`broadcast(fn(_ * _), expected, lumi)`	`draw(Normal(lumi_nom, sigma_lumi))`	—	
`normsys` / `OverallSys`	`broadcast(fn(_ * _), expected, interp_*(lo, 1.0, hi, alpha))`	`draw(Normal(0, 1))`	`interp_poly6_exp`	
`histosys` / `HistoSys`	`interp_*(tmpl_dn, nom, tmpl_up, alpha)`	`draw(Normal(0, 1))`	`interp_poly6_lin`	Replaces nominal directly
`HistoFactor`	same as `histosys`	none (free)	same as `histosys`	Free parameter variant
`shapefactor` / `ShapeFactor`	`broadcast(fn(_ * _), expected, gamma)`	none (free per-bin)	—	`gamma = elementof(cartpow(reals, n_bins))`
`shapesys` / `ShapeSys`	`broadcast(fn(_ * _ / _), nom, gamma, tau)`	`draw(broadcast(Poisson, tau))`	—	`tau = broadcast(fn(pow(_ / _, 2)), nom, sigma)`
`staterror` / `StatError`	`broadcast(fn(_ * _), total_nom, gamma)`	`draw(broadcast(fn(Normal(_, _)), ones, delta))`	—	`delta` from quadrature sum across samples

Notes. Interpolation codes are configurable per modifier; defaults shown. `interp_*` selects the corresponding FlatPPL function. Constraint likelihoods additionally require auxiliary observation models and `likelihoodof` calls (see worked example). Parameter sharing: modifiers with the same name share a single `draw`; the translator must verify compatible constraint types. The `shapesys` row uses the Poisson pseudo-count parameterization (γ has prior mean τ, effective multiplier is γ/τ); an equivalent form uses γ directly as multiplier with a `Gamma(tau + 1, tau)` prior.

11.5.2 Worked example: a HistFactory-style channel

```
# ===== Nominal templates and uncertainties =====
sig_nominal = [12.0, 11.0, 8.0, 5.0]
sig_jes_down = [10.0, 9.5, 7.0, 4.0]
sig_jes_up = [14.0, 12.5, 9.0, 6.0]
bkg_nominal = [50.0, 52.0, 48.0, 45.0]
delta_mc = [0.05, 0.04, 0.06, 0.08]
```

```
# ===== Nuisance parameters (probabilistic constraints) =====
alpha_jes = draw(Normal(0.0, 1.0))
alpha_xsec = draw(Normal(0.0, 1.0))
gamma_stat = draw(broadcast(fn(Normal(_, _)), [1.0, 1.0, 1.0, 1.0], delta_mc))

# ===== Expected counts =====
sig_morphed = interp_poly6_lin(sig_jes_down, sig_nominal, sig_jes_up,
alpha_jes)
kappa_xsec = interp_poly6_exp(0.9, 1.0, 1.1, alpha_xsec)
expected = broadcast(fn(_ * _ * _ + _ * _),
    mu_sig, sig_morphed, kappa_xsec, bkg_nominal, gamma_stat)

# ===== Observation model =====
obs = draw(broadcast(Poisson, expected))

# ===== Likelihood =====
L_obs = likelihoodof(
    lawof(obs, alpha_jes = alpha_jes, alpha_xsec = alpha_xsec,
        gamma_stat = gamma_stat),
    [51, 48, 55, 42])

# ===== Constraint terms =====
aux_jes = draw(Normal(alpha_jes, 1.0))
aux_xsec = draw(Normal(alpha_xsec, 1.0))
aux_stat = draw(broadcast(fn(Normal(_, _)), gamma_stat, delta_mc))

L_constr_jes = likelihoodof(lawof(aux_jes, alpha_jes = alpha_jes), 0.0)
L_constr_xsec = likelihoodof(lawof(aux_xsec, alpha_xsec = alpha_xsec), 0.0)
L_constr_stat = likelihoodof(
    lawof(aux_stat, gamma_stat = gamma_stat), [1.0, 1.0, 1.0, 1.0])

L = joint_likelihood(L_obs, L_constr_jes, L_constr_xsec, L_constr_stat)
```

Key points:

- **Boundary inputs** on `lawof` keep nuisance parameters as kernel parameters rather than marginalizing them — matching HistFactory's product-likelihood structure.
- **Auxiliary observation models** define constraint terms as genuine likelihood functions.
- **`joint_likelihood`** multiplies observation and constraint factors.
- **`broadcast`** is always required for elementwise bin arithmetic.

11.6 Stan profile

Stan is a probabilistic programming language for Bayesian inference, primarily via HMC/NUTS. It specifies models as joint log-densities over parameters and data in a block-structured program (data, parameters, model, generated quantities). The Stan profile is simpler than the HS³/RooFit

profile because Stan models are single joint log-densities with no separate likelihood objects, no measure algebra, and no compositional kernel structure.

11.6.1 Stan → FlatPPL

A Stan model block defines a joint distribution over parameters and observations. The most direct translation maps every ~ statement to a FlatPPL `draw(...)` — both on model parameters and on observed data — producing a joint model:

- Stan ~ statements map to `draw(...)`.
- Stan `target += ...` accumulates contributions to a joint log-density; in FlatPPL this corresponds to `logweighted(...)` applied to the underlying joint measure.
- Stan's parameter block maps to `draw(...)` with appropriate priors.
- Stan's data block defines literal values or `load_data(...)`.
- Stan's transformed parameters/data blocks map to deterministic computation.

The resulting FlatPPL model is a joint distribution that can be decomposed via `disintegrate` (structural disintegration) to extract the forward kernel and prior together, and then combined with observed data via `likelihoodof` — something Stan's block structure does not expose directly.

11.6.2 FlatPPL → Stan

FlatPPL models that express a joint distribution over parameters and observations (without separate likelihood objects) map to Stan. The profile includes:

FlatPPL construct	Stan equivalent
`draw(D(...))`	`x ~ D(...)` (generative fragment)
`elementof(S)`	parameter declaration with constraints
Deterministic computation	transformed parameters / model block
`logweighted(lw, M)`	`target += lw`
`lawof(record(...))`	implicit in block structure

What does not map. Stan does not support:

- Separate likelihood objects (`likelihoodof`, `joint_likelihood`)
- Measure algebra (`weighted`, `superpose`, `joint`, `jointchain`, `chain`, `pushfwd`)
- Explicit density evaluation (`densityof`, `logdensityof`)
- `PoissonProcess` / `BinnedPoissonProcess` as first-class constructs
- Frequentist workflows (profile likelihood ratios, etc.)

Models using these features must be restructured before export to Stan, if feasible.

11.6.3 Stan distribution mapping

The following tables summarize major correspondences; they are illustrative rather than exhaustive.

FlatPPL	Stan	Parameter notes
`Uniform`	`uniform`	`support` → `(alpha, beta)` bounds
`Normal`	`normal`	`mu` → `mu`, `sigma` → `sigma`
`Cauchy`	`cauchy`	`location` → `mu`, `scale` → `sigma`
`StudentT`	`student_t`	`nu` → `nu`; Stan has location-scale form
`Logistic`	`logistic`	`mu` → `mu`, `s` → `sigma`
`LogNormal`	`lognormal`	`mu` → `mu`, `sigma` → `sigma`
`Exponential`	`exponential`	`rate` → `beta` (Stan uses rate)
`Gamma`	`gamma`	`shape` → `alpha`, `rate` → `beta`
`Weibull`	`weibull`	`shape` → `alpha`, `scale` → `sigma`
`InverseGamma`	`inv_gamma`	`shape` → `alpha`, `scale` → `beta`
`Beta`	`beta`	`alpha` → `alpha`, `beta` → `beta`
`Bernoulli`	`bernoulli`	`p` → `theta`
`Categorical`	`categorical`	`p` → `theta`
`Binomial`	`binomial`	`n` → `N`, `p` → `theta`
`Poisson`	`poisson`	`rate` → `lambda`
`MvNormal`	`multi_normal`	`mu` → `mu`, `cov` → `Sigma`
`Wishart`	`wishart`	`nu` → `nu`, `scale` → `S`
`InverseWishart`	`inv_wishart`	`nu` → `nu`, `scale` → `S`
`LKJCholesky`	`lkj_corr_cholesky`	`eta` → `eta`
`Dirichlet`	`dirichlet`	`alpha` → `alpha`
`Multinomial`	`multinomial`	`n` → `N`, `p` → `theta`

11.6.4 Stan function mapping

FlatPPL	Stan	Notes
`exp`, `log`, `sqrt`, `abs`, `sin`, `cos`	same names	
`pow`	`^` operator	
`sum`, `product`	`sum`, `prod`	
`ifelse`	ternary `? :`	
`lower_cholesky`	`cholesky_decompose`	
`det`, `inv`, `trace`	`determinant`, `inverse`, `trace`	

FlatPPL	Stan	Notes
`broadcast`	vectorized operations	Stan auto-vectorizes for standard distributions; general `broadcast` may require explicit loops

11.7 Future profiles

Additional profiles for systems such as Pyro, NumPyro, and PyMC are natural extensions of the same framework.

---

12 Appendix: Implementations

This appendix is a collection of functional equivalents between FlatPPL constructs and existing package ecosystems in programming languages like Python and Julia. The focus is on existing building blocks that can be used for full FlatPPL implementations, not on the runtime or inference machinery that come as part of existing ecosystems. This appendix is not normative, it is meant to list possibilities, not to prescribe particular design choices.

12.1 Target ecosystems

These two ecosystems are likely good candidates to underpin a full FlatPPL implementation, but there are of course many other options:

- **Python/JAX**: JAX provides the computation substrate (array ops, autodiff, JIT, accelerator support via MLIR/StableHLO). Distribution objects are available from `numpyro.distributions` or TensorFlow Probability on JAX (`tfp.substrates.jax`), both usable as standalone libraries independently of their respective PPL runtimes. In turn, functions and distributions expressed in FlatPPL could be made API-compatible with NumPyro and TF Probability, allowing users to leverage the rich inference tools built on top of them.
- **Julia**: MeasureBase.jl provides the measure-theoretic foundation and Distributions.jl (augmented by DistributionsHEP.jl and other packages) provides implementations of many distributions. In turn, functions, distributions, and measures expressed in FlatPPL would fit naturally into the MeasureBase.jl and Distributions.jl APIs.

12.2 Distributions

The table below lists approximate ecosystem equivalents, not exact constructor names. This table may well be incomplete:

FlatPPL	NumPyro	TF Probability	Julia
`Uniform`	`Uniform`	`Uniform`	`Uniform`
`Normal`	`Normal`	`Normal`	`Normal`
`GeneralizedNormal`	—	`GeneralizedNormal`	—

FlatPPL	NumPyro	TF Probability	Julia
`Cauchy`	`Cauchy`	`Cauchy`	`Cauchy`
`StudentT`	`StudentT`	`StudentT`	`TDist`
`Logistic`	`Logistic`	`Logistic`	`Logistic`
`LogNormal`	`LogNormal`	`LogNormal`	`LogNormal`
`Exponential`	`Exponential`	`Exponential`	`Exponential`
`Gamma`	`Gamma`	`Gamma`	`Gamma`
`Weibull`	`Weibull`	`Weibull`	`Weibull`
`InverseGamma`	`InverseGamma`	`InverseGamma`	`InverseGamma`
`Beta`	`Beta`	`Beta`	`Beta`
`Bernoulli`	`Bernoulli`	`Bernoulli`	`Bernoulli`
`Categorical`	`Categorical`	`Categorical`	`Categorical`
`Binomial`	`Binomial`	`Binomial`	`Binomial`
`Poisson`	`Poisson`	`Poisson`	`Poisson`
`ContinuedPoisson`	—	—	—
`MvNormal`	`MultivariateNormal`	`MultivariateNormalTriL`	`MvNormal`
`Wishart`	`WishartCholesky` (via TFP)	`WishartTriL`	`Wishart`
`InverseWishart`	`InverseWishart` (via TFP)	`InverseWishart`	`InverseWishart`
`LKJ`	`LKJ`	`LKJ`	`LKJ`
`LKJCholesky`	`LKJCholesky`	`CholeskyLKJ`	`LKJCholesky`
`Dirichlet`	`Dirichlet`	`Dirichlet`	`Dirichlet`
`Multinomial`	`Multinomial`	`Multinomial`	`Multinomial`
`PoissonProcess`	—	—	—
`BinnedPoissonProcess`	—	—	—
`CrystalBall`	—	—	`CrystalBall` (DistributionsHEP.jl)
`DoubleSidedCrystalBall`	—	—	`DoubleCrystalBall` (DistributionsHEP.jl)
`Argus`	—	—	`ArgusBG` (DistributionsHEP.jl)
`BreitWigner`	`Cauchy`	`Cauchy`	`Cauchy`
`RelativisticBreitWigner`	—	—	—
`Voigtian`	—	—	—

FlatPPL	NumPyro	TF Probability	Julia
`BifurcatedGaussian`	—	—	`BifurcatedGaussian` (DistributionsHEP.jl)

13 Declaration of generative AI in the writing process

During the preparation of this work, the authors used various LLM-based systems to assist with structural organization, improving exposition, drafting and refinement of the manuscript prose and copyediting. The underlying concepts and ideas presented in this document, as well as the original content drafts, are the work of the human authors. The authors reviewed and edited all AI-assisted output and take full responsibility for the final content, accuracy, and integrity of the document.

14 References

BAT.jl — Bayesian Analysis Toolkit in Julia. https://github.com/bat/BAT.jl

Birch — A universal probabilistic programming language. https://birch-lang.org/

Böck, M., Schröder, A., Cito, J. (2024). LASAPP: Language-agnostic static analysis for probabilistic programs. ASE '24. https://doi.org/10.1145/3691620.3695031

Böck, M., Cito, J. (2025). Static factorisation of probabilistic programs. OOPSLA 2026. https://arxiv.org/abs/2508.20922

Carpenter, B. et al. (2017). Stan: A probabilistic programming language. *J. Stat. Softw.* 76(1). https://mc-stan.org/

DensityInterface.jl. https://github.com/JuliaMath/DensityInterface.jl

Fenske, T., Popko, A., Bader, S., Kirste, T. (2025). Representation-agnostic probabilistic programming. https://arxiv.org/abs/2512.23740

Fowlie, A. (2025). stanhf: HistFactory models in Stan. *Eur. Phys. J. C* 85:923. https://arxiv.org/abs/2503.22188

Giry, M. (1982). A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, LNM 915:68–85. https://ncatlab.org/nlab/show/Giry+monad

Gorinova, M. I., Gordon, A. D., Sherlock, C. (2019). Probabilistic programming with densities in SlicStan. *Proc. ACM Program. Lang.* 3(POPL):35.

GraphPPL.jl. https://github.com/reactivebayes/GraphPPL.jl

HS³ — HEP Statistics Serialization Standard. https://hep-statistics-serialization-standard.github.io/ · GitHub: https://github.com/hep-statistics-serialization-standard

Keras Functional API. https://keras.io/guides/functional_api/

Narayanan, P. et al. (2016). Probabilistic inference by program transformation in Hakaru. FLOPS. https://github.com/hakaru-dev/hakaru

pyhf — pure-Python HistFactory implementation. https://github.com/scikit-hep/pyhf

pyhs3 — Python HS³ implementation. https://pypi.org/project/pyhs3/

PyTensor (formerly Aesara) — graph cloning and symbolic computation. https://pytensor.readthedocs.io/

RooFit — Statistical modeling toolkit in ROOT. https://root.cern/manual/roofit/

RxInfer.jl — Reactive message-passing inference. https://github.com/ReactiveBayes/RxInfer.jl

Shan, C., Ramsey, N. (2017). Exact Bayesian inference by symbolic disintegration. *J. Funct. Program.*

Staton, S. et al. (2016). Semantics for probabilistic programming. LICS. https://arxiv.org/abs/1601.04943

Staton, S. (2017). Commutative semantics for probabilistic programming. ESOP.

Weiser, M. (1981). Program slicing. *Proc. 5th ICSE.*