Module `Ppl`

A shallowly embedded DSL for Probabilistic Programming

Universal PPL in OCaml

Core

`module Dist : sig ... end`
Module used for defining probabilistic models

`module Primitive : sig ... end`
Module defining a type for primitive distributions

`module Empirical : sig ... end`
A module for empirical distributions generated from samplers

Extra

`module Plot : sig ... end`
Plotting utilies

`module Evaluation : sig ... end`
A module for evaluating the correctness of models and inference procedures

`module Inference : sig ... end`
Implementation of inference algorithms

`module Helpers : sig ... end`
Utilities for working with distributions

Module `Ppl.Dist`

Module used for defining probabilistic models

Contains a type `dist` which is used to represent probabilistic models.

`exception Undefined`

`module Prob : Ppl__.Sigs.Prob`
The module used to represent probability, can be switched to use log probs

```
type prob = Prob.t
```
A type for which values need to sum to 1 (not an enforced property)

```
type likelihood = Prob.t
```
A type for which values don't need to sum to 1 (not an enforced property)

```
type 'a samples = ('a * prob) list
```
A set of weighted samples, summing to one

```
type _ dist = private
```

`| Return : 'a -> 'a dist`	distribution with a single value
`| Bind : 'a dist * ('a -> 'b dist) -> 'b dist`	monadic bind
`| Primitive : 'a Primitive.t -> 'a dist`	primitive exact distribution
`| Conditional : ('a -> likelihood) * 'a dist -> 'a dist`	variant that defines likelihood model
`| Independent : 'a dist * 'b dist -> ('a * 'b) dist`	for combining two independent distributions

GADT for representing distributions, private to avoid direct manipulation

Condition Operators

```
val condition' : ('a -> float) -> 'a dist -> 'a dist
```
most general condition operator

```
val condition : bool -> 'a dist -> 'a dist
```
hard conditioning

```
val score : float -> 'a dist -> 'a dist
```
soft conditioning

```
val observe : 'a -> 'a Primitive.t -> 'b dist -> 'b dist
```
soft conditioning for observations from a known distribution

Monad Functions

Monad functions

```
type 'a t

val return : 'a -> 'a t
val bind : 'a t -> ('a -> 'b t) -> 'b t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
val let* : 'a t -> ('a -> 'b t) -> 'b t
val fmap : ('a -> 'b) -> 'a t -> 'b t
val liftM : ('a -> 'b) -> 'a t -> 'b t
val liftM2 : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
val mapM : ('a -> 'b t) -> 'a list -> 'b list t
val sequence : 'a t list -> 'a list t

val and* : 'a dist -> 'b dist -> ('a * 'b) dist
```

Primitives

These functions create dist values which correspond to primitive distributions so that they can be used in models.

```
type 'a primitive

val binomial : int -> float -> int primitive
```
Create a binomial distribution, the output is the number of successes from n independent trials with probability of success p

```
val normal : float -> float -> float primitive
val categorical : ('a * float) list -> 'a primitive
val discrete_uniform : 'a list -> 'a primitive
val beta : float -> float -> float primitive
val gamma : float -> float -> float primitive
val continuous_uniform : float -> float -> float primitive

val bernoulli : float -> bool dist
```

Sampling

```
val sample : 'a dist -> 'a
val sample_n : int -> 'a dist -> 'a array
val sample_with_score : 'a dist -> 'a * likelihood
```

Prior Distribution

```
val prior' : 'a dist -> 'a dist

val prior_with_score : 'a dist -> ('a * likelihood) dist
val support : 'a dist -> 'a list

module PplOps : Ppl__.Sigs.Ops with type 'a dist := 'a dist
```
Common operators for combining distributions

Module Ppl.Inference

Implementation of inference algorithms

Inference algorithms to be called on probabilistic models defined using Dist


```
exception Undefined
```

```
type 'a samples = ('a * Dist.prob) list
```

Helpers

```
val unduplicate : 'a samples -> 'a samples
val resample : 'a samples -> 'a samples Dist.dist
val normalise : 'a samples -> 'a samples
val flatten : ('a samples * Dist.prob) list -> 'a samples
```

Exact Inference

```
val enumerate : 'a Dist.dist -> float -> 'a samples
val exact_inference : 'a Dist.dist -> 'a Dist.dist
```

Importance Sampling

```
val importance : int -> 'a Dist.dist -> 'a samples Dist.dist
```

Rejetion Sampling

```
type rejection_type =

| Hard
| Soft

val pp_rejection_type : Stdlib.Format.formatter -> rejection_type -> unit
val show_rejection_type : rejection_type -> string
val rejection : ? n:int -> rejection_type -> 'a Dist.dist -> 'a Dist.dist
```

Sequential Monte Carlo

```
val smcStandard' : int -> 'a Dist.dist -> 'a Dist.dist
val smcMultiple' : int -> int -> 'a Dist.dist -> 'a Dist.dist
```

Metropolis Hastings

```
val mh' : int -> 'a Dist.dist -> 'a Dist.dist
```

Particle Independent Metropolis Hastings

```
val pimh' : int -> int -> 'a Dist.dist -> 'a Dist.dist
```

Particle Cascade

```
val cascade' : int -> 'a Dist.dist -> 'a Dist.dist
```

Common

```
type infer_strat =

| MH of int
| SMC of int
| PC of int
```

```
| PIMH of int
| Importance of int
| Rejection of int * rejection_type
| Prior
| Enum
| Forward

val pp_infer_strat : Stdlib.Format.formatter -> infer_strat -> unit

val infer : 'a Dist.dist -> infer_strat -> 'a Dist.dist
val infer_sampler : 'a Dist.dist -> infer_strat -> unit -> 'a
```


Module `Ppl.Empirical`

A module for empirical distributions generated from samplers

Contains a signature as well as two implementations, for continuous and discrete distributions respectively

```
module type S = sig ... end
module Discrete : S
module ContinuousArr : sig ... end
```


Module type `Empirical.S`

```
type 'a t

val from_dist : ? n:int -> 'a Dist.dist -> 'a t
```
Create a empirical distribution from a distribution object, using n samples to approximate it

```
val empty : 'a t
```
Create an empty distribution

```
val add_sample : 'a t -> 'a -> 'a t
```
Add another sample to the distribution

```
val get_num : 'a t -> 'a -> int
```
Get the numer of samples with the value

```
val get_prob : 'a t -> 'a -> float
```
Get the probability of a particular value

```
val to_pdf : 'a t -> 'a -> float
```
Create a pdf function

```
val print_map : (module Core.Pretty_printer.S with type t = 'a) -> 'a t
-> unit
```
print the entire distribution

```
val to_arr : 'a t -> ('a * int) array
```
Get array of samples

```
val support : 'a t -> 'a list
```

Get the set of values for the distribution