webppl Documentation

Release 0.9.10

the webppl contributors

Sep 12, 2017

Contents

CHAPTER 1

Getting Started

Try WebPPL

The quickest way to get started using WebPPL is by running programs using the code box on webppl.org. WebPPL can also be *installed locally* and run from the command line.

Learning

If you're new to probabilistic programming, Probabilistic Models of Cognition is a great place to start learning the paradigm.

The best guide to using WebPPL is The Design and Implementation of Probabilistic Programming Languages. The examples will also be helpful in learning the syntax.

Need help?

If you have any questions about installing WebPPL or need help with your code, you can get help on the Google group.

CHAPTER 2

Language Overview

The WebPPL language begins with a subset of JavaScript, and adds to it *primitive distributions* and operations to perform *sampling*, *conditioning* and *inference*.

Syntax

Following the notation from the Mozilla Parser API, the language consists of the subset of JavaScript that can be built from the following syntax elements, each shown with an `example`:

Element	Example
Program	A complete program, consisting of a sequence of statements
BlockStatement	A sequence of statements surrounded by braces, `{var x = 1; var y = 2;}`
ExpressionStatement	A statement containing a single expression, `3 + 4;`
ReturnStatement	`return 3;`
EmptyStatement	A solitary semicolon, `;`
IfStatement	`if (x > 1) { return 1; } else { return 2; }`
VariableDeclaration	`var x = 5;`
Identifier	`x`
Literal	`3`
FunctionExpression	`function (x) { return x; }`
CallExpression	`f(x)`
ConditionalExpression	`x ? y : z`
ArrayExpression	`[1, 2, 3]`
MemberExpression	`Math.log`
BinaryExpression	`3 + 4`
LogicalExpression	`true || false`
UnaryExpression	`-5`
ObjectExpression	`{a: 1, b: 2}`
AssignmentExpression	`globalStore.a = 1` (Assignment is only supported by the *global store*.)

Note that general assignment expressions and looping constructs are not currently supported (e.g. `for`, `while`, `do`). This is because a purely functional language is much easier to transform into Continuation-Passing Style (CPS),

which the WebPPL implementation uses to implement inference algorithms such as *enumeration* and *SMC*. While these restrictions mean that common JavaScript programming patterns aren't possible, this subset is still universal, because we allow recursive and higher-order functions. It encourages a functional style, similar to Haskell or LISP, that is pretty easy to use (once you get used to thinking functionally!).

Here is a (very boring) program that uses much of the available syntax:

```
var foo = function(x) {
  var bar = Math.exp(x);
  var baz = x === 0 ? [] : [Math.log(bar), foo(x-1)];
  return baz;
}

foo(5);
```

Calling JavaScript Functions

JavaScript functions can be called from a WebPPL program, with a few restrictions:

1. JavaScript functions must be deterministic and cannot carry state from one call to another. (That is, the functions must be 'referentially transparent': calling `obj.foo(args)` must always return the same value when called with given arguments.)
2. JavaScript functions can't be called with a WebPPL function as an argument (that is, they can't be higher-order).
3. JavaScript functions must be invoked as the method of an object (indeed, this is the only use of object method invocation currently possible in WebPPL).

All of the JavaScript functions built into the environment in which WebPPL is running are automatically available for use. Additional functions can be added to the environment through the use of *packages*.

Note that since JavaScript functions must be called as methods on an object, it is not possible to call global JavaScript functions such as `parseInt()` directly. Instead, such functions should be called as methods on the built-in object `_top`. e.g. `_top.parseInt('0')`.

CHAPTER 3

Installation

First, install git.

Second, install Node.js. WebPPL is written in JavaScript, and requires Node to run. After it is installed, you can use npm (**n**ode **p**ackage **m**anager) to install WebPPL:

```
npm install -g webppl
```

Create a file called `test.wppl`:

```
var greeting = function () {
    return flip(.5) ? "Hello" : "Howdy"
}

var audience = function () {
    return flip(.5) ? "World" : "Universe"
}

var phrase = greeting() + ", " + audience() + "!"

phrase
```

Run it with this command:

```
webppl test.wppl
```

Updating

WebPPL is in active development. To update WebPPL, run:

```
npm update -g webppl
```

CHAPTER 4

Usage

Running WebPPL programs:

```
webppl examples/geometric.wppl
```

Arguments

Requiring Node.js core modules or *WebPPL packages*:

```
webppl model.wppl --require fs
webppl model.wppl --require webppl-viz
```

Seeding the random number generator:

```
webppl examples/lda.wppl --random-seed 2344512342
```

Compiling WebPPL programs to JavaScript:

```
webppl examples/geometric.wppl --compile --out geometric.js
```

The compiled file can be run using nodejs:

```
node geometric.js
```

Passing arguments to the program

Command line arguments can be passed through to the WebPPL program by placing them after a single `--` argument. Such arguments are parsed (with minimist) and the result is bound to the global variable `argv`.

For example, this program:

```
display(argv);
```

When run with:

```
webppl model.wppl -- --my-flag --my-num 100 --my-str hello
```

Will produce the following output:

```
{ _: ['model.wppl'], 'my-flag': true, 'my-num': 100, 'my-str': 'hello' }
```

Once compiled, a program takes its arguments directly, i.e. the `--` separator is not required.

CHAPTER 5

Debugging

WebPPL provides error messages that try to be informative. In addition there is debugging software you can use for WebPPL programs.

To debug WebPPL programs running in Chrome, enable pause on JavaScript exceptions in the Chrome debugger.

To debug WebPPL programs running in nodejs, use node debugger as follows:

1. Add `debugger;` statements to `my-program.wppl` to indicate breakpoints.
2. Run your compiled program in debug mode:

   ```
   node debug path/to/webppl my-program.wppl
   ```

   Note that you will need the full path to the `webppl` executable. This might be in the `lib` folder of your `node` directory if you installed with `npm`. On many systems you can avoid entering the path manually by using the following command:

   ```
   node debug `which webppl` my-program.wppl
   ```

3. To navigate to your breakpoint within the debugger interface, type `cont` or `c`. At any break point, you can type `repl` to interact with the variables. Here's some documentation for this debugger.

CHAPTER 6

Sample

Guides

A number of *inference* strategies make use of an auxiliary distribution which we call a *guide distribution*. They are specified like so:

```
sample(dist, {guide: guideFn});
```

Where `guideFn` is a function that takes zero arguments, and returns a distribution object.

For example:

```
sample(Cauchy(params), {
  guide: function() {
    return Gaussian(guideParams);
  }
});
```

Note that such functions will only be called when using an inference strategy that makes use of the guide.

In some situations, it is convenient to be able to specify part of a guide computation outside of the functions passed to `sample`. This can be accomplished with the `guide` function, which takes a function of zero arguments representing the computation:

```
guide(function() {
  // Some guide computation.
});
```

As with the functions passed to `sample`, the function passed to `guide` will only be called when required for inference.

It's important to note that `guide` does not return the value of the computation. Instead, the *global store* should be used to pass results to subsequent guide computations. This arrangement encourages a programming style in which there is separation between the model and the guide.

Default Guide Distributions

Both *optimization* and *forward sampling* from the guide require that all *random choices* in the model have a corresponding *guide distribution*. So, for convenience, these methods automatically use an appropriate *default guide distribution* at any random choice in the model for which a guide distribution is not specified explicitly.

Default guide distributions can also be used with *SMC*. See the documentation for the `importance` option for details.

The default guide distribution used at a particular random choice:

- Is independent of all other guide distributions in the program.
- Has its type determined by the type of the distribution specified in the model for the random choice.
- Has each of its continuous parameters hooked up to an optimizable *parameter*. These parameters are not shared with any other guide distributions in the program.

For example, the default guide distribution for a `Bernoulli` random choice could be written explicitly as:

```
var x = sample(Bernoulli({p: 0.5}), {guide: function() {
  return Bernoulli({p: Math.sigmoid(param())});
}});
```

Drift Kernels

Introduction

The default behavior of MH based *inference* algorithms is to generate proposals by sampling from the prior. This strategy is generally applicable, but can be inefficient when the prior places little mass in areas where the posterior mass in concentrated. In such situations the algorithm may make many proposals before a move is accepted.

An alternative is to sample proposals from a distribution centered on the previous value of the random choice to which we are proposing. This produces a random walk that allows inference to find and explore areas of high probability in a more systematic way. This type of proposal distribution is called a drift kernel.

This strategy has the potential to perform better than sampling from the prior. However, the width of the proposal distribution affects the efficiency of inference, and will often need tuning by hand to obtain good results.

Specifying drift kernels

A drift kernel is represented in a WebPPL program as a function that maps from the previous value taken by a random choice to a *distribution*.

For example, to propose from a Gaussian distribution centered on the previous value we can use the following function:

```
var gaussianKernel = function(prevVal) {
  return Gaussian({mu: prevVal, sigma: .1});
};
```

This function can be used to specify a drift kernel at any `sample` statement using the `driftKernel` option like so:

```
sample(dist, {driftKernel: kernelFn});
```

To use our `gaussianKernel` with a Cauchy random choice we would write:

```
sample(Cauchy(params), {driftKernel: gaussianKernel});
```

Helpers

A number of built-in helpers provide sensible drift kernels for frequently used distributions. These typically take the same parameters as the *distribution* from which they sample, plus an extra parameter to control the width of the proposal distribution.

gaussianDrift (*{mu: ..., sigma: ..., width: ...}*)

dirichletDrift (*{alpha: ..., concentration: ...}*)

uniformDrift (*{a: ..., b: ..., width: ...}*)

A generative process is described in WebPPL by combining samples drawn from *distribution objects* with deterministic computation. Samples are drawn using the primitive `sample` operator like so:

```
sample(dist);
```

Where `dist` is either a *primitive distribution* or a distribution obtained as the result of *marginal inference*.

For example, a sample from a standard Gaussian distribution can be generated using:

```
sample(Gaussian({mu: 0, sigma: 1}));
```

For convenience, all *primitive distributions* have a corresponding helper function that draws a sample from that distribution. For example, sampling from the standard Gaussian can be more compactly written as:

```
gaussian({mu: 0, sigma: 1});
```

The name of each of these helper functions is obtained by taking the name of the corresponding distribution and converting the first letter to lower case.

The `sample` primitive also takes an optional second argument. This is used to specify *guide distributions* and *drift kernels*.

CHAPTER 7

Distributions

Distribution objects represent probability distributions, they have two principle uses:

1. Samples can be generated from a distribution by passing a distribution object to the *sample* operator.
2. The logarithm of the probability (or density) that a distribution assigns to a value can be computed using `dist.score(val)`. For example:

```
Bernoulli({p: .1}).score(true); // returns Math.log(.1)
```

Several *primitive distributions* are built into the language. Further distributions are created by performing *marginal inference*.

Primitives

`Bernoulli` (*{p: ...}*)

- p: success probability *(real [0, 1])*

Distribution over `{true, false}`

Wikipedia entry

`Beta` (*{a: ..., b: ...}*)

- a: shape *(real (0, Infinity))*
- b: shape *(real (0, Infinity))*

Distribution over `[0, 1]`

Wikipedia entry

`Binomial` (*{p: ..., n: ...}*)

- p: success probability *(real [0, 1])*
- n: number of trials *(int (>=1))*

Distribution over the number of successes for `n` independent `Bernoulli({p: p})` trials.

Wikipedia entry

`Categorical` (*{ps: ..., vs: ...}*)

- ps: probabilities (can be unnormalized) *(vector or real array [0, Infinity))*
- vs: support *(any array)*

Distribution over elements of `vs` with `P(vs[i])` proportional to `ps[i].ps` may be omitted, in which case a uniform distribution over `vs` is returned.

Wikipedia entry

`Cauchy` (*{location: ..., scale: ...}*)

- location: *(real)*
- scale: *(real (0, Infinity))*

Distribution over `[-Infinity, Infinity]`

Wikipedia entry

`Delta` (*{v: ...}*)

- v: support element *(any)*

Discrete distribution that assigns probability one to the single element in its support. This is only useful in special circumstances as sampling from `Delta({v: val})` can be replaced with `val` itself. Furthermore, a `Delta` distribution parameterized by a random choice should not be used with MCMC based inference, as doing so produces incorrect results.

`DiagCovGaussian` (*{mu: ..., sigma: ...}*)

- mu: mean *(tensor)*
- sigma: standard deviations *(tensor (0, Infinity))*

A distribution over tensors in which each element is independent and Gaussian distributed, with its own mean and standard deviation. i.e. A multivariate Gaussian distribution with diagonal covariance matrix. The distribution is over tensors that have the same shape as the parameters `mu` and `sigma`, which in turn must have the same shape as each other.

`Dirichlet` (*{alpha: ...}*)

- alpha: concentration *(vector (0, Infinity))*

Distribution over probability vectors. If `alpha` has length `d` then the distribution is over probability vectors of length `d`.

Wikipedia entry

`Discrete` (*{ps: ...}*)

- ps: probabilities (can be unnormalized) *(vector or real array [0, Infinity))*

Distribution over `{0,1,...,ps.length-1}` with P(i) proportional to `ps[i]`

Wikipedia entry

`Exponential` (*{a: ...}*)

- a: rate *(real (0, Infinity))*

Distribution over `[0, Infinity]`

Wikipedia entry

Gamma (*{shape: ..., scale: ...}*)

•shape: *(real (0, Infinity))*

•scale: *(real (0, Infinity))*

Distribution over positive reals.

Wikipedia entry

Gaussian (*{mu: ..., sigma: ...}*)

•mu: mean *(real)*

•sigma: standard deviation *(real (0, Infinity))*

Distribution over reals.

Wikipedia entry

KDE (*{data: ..., width: ...}*)

•data: data array

•width: kernel width

A distribution based on a kernel density estimate of `data`. A Gaussian kernel is used, and both real and vector valued data are supported. When the data are vector valued, `width` should be a vector specifying the kernel width for each dimension of the data. When `width` is omitted, Silverman's rule of thumb is used to select a kernel width. This rule assumes the data are approximately Gaussian distributed. When this assumption does not hold, a `width` should be specified in order to obtain sensible results.

Wikipedia entry

Laplace (*{location: ..., scale: ...}*)

•location: *(real)*

•scale: *(real (0, Infinity))*

Distribution over `[-Infinity, Infinity]`

Wikipedia entry

LogisticNormal (*{mu: ..., sigma: ...}*)

•mu: mean *(vector)*

•sigma: standard deviations *(vector (0, Infinity))*

A distribution over probability vectors obtained by transforming a random variable drawn from `DiagCovGaussian({mu: mu, sigma: sigma})`. If `mu` and `sigma` have length `d` then the distribution is over probability vectors of length `d+1`.

Wikipedia entry

LogitNormal (*{mu: ..., sigma: ..., a: ..., b: ...}*)

•mu: location *(real)*

•sigma: scale *(real (0, Infinity))*

•a: lower bound *(real)*

•b: upper bound (>a) *(real)*

A distribution over `(a,b)` obtained by scaling and shifting a standard logit-normal.

Wikipedia entry

Mixture (*{dists: ..., ps: ...}*)

- dists: array of component distributions
- ps: component probabilities (can be unnormalized) *(vector or real array [0, Infinity))*

A finite mixture of distributions. The component distributions should be either all discrete or all continuous. All continuous distributions should share a common support.

Multinomial (*{ps: ..., n: ...}*)

- ps: probabilities *(real array with elements that sum to one)*
- n: number of trials *(int (>=1))*

Distribution over counts for `n` independent `Discrete({ps: ps})` trials.

Wikipedia entry

MultivariateBernoulli (*{ps: ...}*)

- ps: probabilities *(vector [0, 1])*

Distribution over a vector of independent Bernoulli variables. Each element of the vector takes on a value in `{0, 1}`. Note that this differs from `Bernoulli` which has support `{true, false}`.

MultivariateGaussian (*{mu: ..., cov: ...}*)

- mu: mean *(vector)*
- cov: covariance *(positive definite matrix)*

Multivariate Gaussian distribution with full covariance matrix. If `mu` has length d and `cov` is a `d`-by-`d` matrix, then the distribution is over vectors of length `d`.

Wikipedia entry

Poisson (*{mu: ...}*)

- mu: mean *(real (0, Infinity))*

Distribution over integers.

Wikipedia entry

RandomInteger (*{n: ...}*)

- n: number of possible values *(int (>=1))*

Uniform distribution over `{0,1,...,n-1}`

Wikipedia entry

TensorGaussian (*{mu: ..., sigma: ..., dims: ...}*)

- mu: mean *(real)*
- sigma: standard deviation *(real (0, Infinity))*
- dims: dimension of tensor *(int (>=1) array)*

Distribution over a tensor of independent Gaussian variables.

TensorLaplace (*{location: ..., scale: ..., dims: ...}*)

- location: *(real)*
- scale: *(real (0, Infinity))*
- dims: dimension of tensor *(int (>=1) array)*

Distribution over a tensor of independent Laplace variables.

Uniform(*{a: ..., b: ...}*)

- a: lower bound *(real)*
- b: upper bound (>a) *(real)*

Continuous uniform distribution over `[a, b]`

Wikipedia entry

CHAPTER 8

Inference

Marginal inference (or just *inference*) is the process of reifying the distribution on return values implicitly represented by a *stochastic computation*.

(In general, computing this distribution is intractable, so often the goal is to compute an approximation to it.)

This is achieved in WebPPL using the `Infer` function, which takes a function of zero arguments representing a stochastic computation and returns the distribution on return values represented as a *distribution object*. For example:

```
Infer(function() {
    return flip() + flip();
});
```

This example has no inference options specified. By default, `Infer` will perform inference using one of the methods among enumeration, rejection sampling, SMC and MCMC. The method to use is chosen by a decision tree based on the characteristics of the given model, such as whether it is enumerable in a timely manner, whether there are interleaving samples and factors etc. Several other implementations of marginal inference are also built into WebPPL. Information about the individual methods is available here:

Methods

Enumeration

`Infer` (*{model: ..., method: 'enumerate'[, ...]}*)

This method performs inference by enumeration.

The following options are supported:

`maxExecutions`

Maximum number of (complete) executions to enumerate.

Default: `Infinity`

strategy
The traversal strategy used to explore executions. Either `'likelyFirst'`, `'depthFirst'` or `'breadthFirst'`.

Default: `'likelyFirst'` if `maxExecutions` is finite, `'depthFirst'` otherwise.

Example usage:

```
Infer({method: 'enumerate', maxExecutions: 10, model: model});
Infer({method: 'enumerate', strategy: 'breadthFirst', model: model});
```

Rejection sampling

Infer (*{model: ..., method: 'rejection'[, ...]}*)
This method performs inference using rejection sampling.

The following options are supported:

samples
The number of samples to take.

Default: `100`

maxScore
An upper bound on the total factor score per-execution.

Default: `0`

incremental
Enable incremental mode.

Default: `false`

Incremental mode improves efficiency by rejecting samples before execution reaches the end of the program where possible. This requires every call to `factor(score)` in the program (across all possible executions) to have `score <= 0`.

Example usage:

```
Infer({method: 'rejection', samples: 100, model: model});
```

MCMC

Infer (*{model: ..., method: 'MCMC'[, ...]}*)
This method performs inference using Markov chain Monte Carlo.

The following options are supported:

samples
The number of samples to take.

Default: `100`

lag
The number of additional iterations to perform between samples.

Default: `0`

burn
The number of additional iterations to perform before collecting samples.

Default: `0`

kernel
The transition kernel to use for inference. See *Kernels*.

Default: `'MH'`

verbose
When `true`, print the current iteration and acceptance ratio to the console during inference.

Default: `false`

onlyMAP
When `true`, only the sample with the highest score is retained. The marginal is a delta distribution on this value.

Default: `false`

Example usage:

```
Infer({method: 'MCMC', samples: 1000, lag: 100, burn: 5, model: model});
```

Kernels

The following kernels are available:

MH
Implements single site Metropolis-Hastings. *[wingate11]*

This kernel makes use of any *drift kernels* specified in the model.

Example usage:

```
Infer({method: 'MCMC', kernel: 'MH', model: model});
```

HMC
Implements Hamiltonian Monte Carlo. *[neal11]*

As the HMC algorithm is only applicable to continuous variables, `HMC` is a cycle kernel which includes a MH step for discrete variables.

The following options are supported:

steps
The number of steps to take per-iteration.

Default: `5`

stepSize
The size of each step.

Default: `0.1`

Example usage:

```
Infer({method: 'MCMC', kernel: 'HMC', model: model});
Infer({method: 'MCMC', kernel: {HMC: {steps: 10, stepSize: 1}}, model: model});
```

Incremental MH

Infer(*{model: ..., method: 'incrementalMH'[, ...]}*)

This method performs inference using C3. *[ritchie15]*

This method makes use of any *drift kernels* specified in the model.

The following options are supported:

samples
The number of samples to take.

Default: `100`

lag
The number of additional iterations to perform between samples.

Default: `0`

burn
The number of additional iterations to perform before collecting samples.

Default: `0`

verbose
When `true`, print the current iteration to the console during inference.

Default: `false`

onlyMAP
When `true`, only the sample with the highest score is retained. The marginal is a delta distribution on this value.

Default: `false`

Example usage:

```
Infer({method: 'incrementalMH', samples: 100, lag: 5, burn: 10, model: model});
```

To maximize efficiency when inferring marginals over multiple variables, use the `query` table, rather than building up a list of variable values:

```
var model = function() {
  var hmm = function(n, obs) {
    if (n === 0) return true;
    else {
      var prev = hmm(n-1, obs);
      var state = transition(prev);
      observation(state, obs[n]);
      query.add(n, state);
      return state;
    }
  };
  hmm(100, observed_data);
  return query;
}
Infer({method: 'incrementalMH', samples: 100, lag: 5, burn: 10, model: model});
```

`query` is a write-only table which can be returned from a program (and thus marginalized). The only operation it supports is adding named values:

`query`.**add**(*name*, *value*)

Arguments

- **name** (any) – Name of value to be added to query. Will be converted to string, as JavaScript object keys are.
- **value** (any) – Value to be added to query.

Returns undefined

SMC

Infer (*{model: ..., method: 'SMC'[, ...]}*)

This method performs inference using sequential Monte Carlo. When `rejuvSteps` is 0, this method is also known as a particle filter.

The following options are supported:

particles
The number of particles to simulate.

Default: `100`

rejuvSteps
The number of MCMC steps to apply to each particle at each `factor` statement. With this addition, this method is often called a particle filter with rejuvenation.

Default: `0`

rejuvKernel
The MCMC kernel to use for rejuvenation. See *Kernels*.

Default: `'MH'`

importance
Controls the importance distribution used during inference.

Specifying an importance distribution can be useful when you know something about the posterior distribution, as specifying an importance distribution that is closer to the posterior than the prior will improve the statistical efficiency of inference.

This option accepts the following values:
- `'default'`: When a random choice has a *guide distribution* specified, use that as the importance distribution. For all other random choices, use the prior.
- `'ignoreGuide'`: Use the prior as the importance distribution for all random choices.
- `'autoGuide'`: When a random choice has a *guide distribution* specified, use that as the importance distribution. For all other random choices, use a *default guide distribution* as the importance distribution.

Default: `'default'`

onlyMAP
When `true`, only the sample with the highest score is retained. The marginal is a delta distribution on this value.

Default: `false`

Example usage:

```
Infer({method: 'SMC', particles: 100, rejuvSteps: 5, model: model});
```

Optimization

Infer (*{model: ..., method: 'optimize'[, ...]}*)

This method performs inference by *optimizing* the parameters of the guide program. The marginal distribution is a histogram constructed from samples drawn from the guide program using the optimized parameters.

The following options are supported:

samples
The number of samples used to construct the marginal distribution.

Default: `100`

onlyMAP
When `true`, only the sample with the highest score is retained. The marginal is a delta distribution on this value.

Default: `false`

In addition, all of the options supported by *Optimize* are also supported here.

Example usage:

```
Infer({method: 'optimize', samples: 100, steps: 100, model: model});
```

Forward Sampling

Infer (*{model: ..., method: 'forward'[, ...]}*)

This method builds a histogram of return values obtained by repeatedly executing the program given by `model`, ignoring any `factor` statements encountered while doing so. Since `condition` and `observe` are written in terms of `factor`, they are also effectively ignored.

This means that unlike all other methods described here, forward sampling does not perform marginal inference. However, sampling from a model without any factors etc. taken into account is often useful in practice, and this method is provided as a convenient way to achieve that.

The following options are supported:

samples
The number of samples to take.

Default: `100`

guide
When `true`, sample random choices from the guide. A *default guide distribution* is used for random choices that do not have a guide distribution specified explicitly.

When `false`, sample from the model.

Default: `false`

onlyMAP
When `true`, only the sample with the highest score is retained. The marginal is a delta distribution on this value.

Default: `false`

Example usage:

```
Infer({method: 'forward', model: model});
Infer({method: 'forward', guide: true, model: model});
```

Bibliography

Conditioning

Conditioning is supported through the use of the `condition`, `observe` and `factor` operators. Only a brief summary of these methods is given here. For a more detailed introduction, see the Probabilistic Models of Cognition chapter on conditioning.

Note that because these operators *interact* with inference, they can only be used *during* inference. Attempting to use them outside of inference will produce an error.

`condition`(*bool*)

Conditions the marginal distribution on an arbitrary proposition. Here, `bool` is the value obtained by evaluating the proposition.

Example usage:

```
var model = function() {
  var a = flip();
  var b = flip();
  condition(a || b)
  return a;
};
```

`observe`(*distribution*, *value*[, *sampleOpts*])

Conceptually, this is shorthand for drawing a value from `distribution` and then conditioning on the value drawn being equal to `value`, which could be written as:

```
var x = sample(distribution);
condition(x === value);
return x;
```

However, in many cases expressing the condition in this way would be exceedingly inefficient, so `observe` uses a more efficient implementation internally.

In particular, it's *essential* to use `observe` to condition on the value drawn from a *continuous* `distribution`.

When `value` is `undefined` no conditioning takes place, and `observe` simply returns a sample from `distribution`. In this case, `sampleOpts` can be used to specify any options that should be used when sampling. Valid options are exactly those that can be given as the second argument to *sample*.

Example usage:

```
var model = function() {
  var mu = gaussian(0, 1);
  observe(Gaussian({mu: mu, sigma: 1}), 5);
  return mu;
};
```

`factor`(*score*)

Adds `score` to the log probability of the current execution.

CHAPTER 9

Optimization

Optimization provides an alternative approach to *marginal inference*.

In this section we refer to the program for which we would like to obtain the marginal distribution as the *target program*.

If we take a target program and add a *guide distribution* to each random choice, then we can define the *guide program* as the program you get when you sample from the guide distribution at each `sample` statement and ignore all `factor` statements.

If we endow this guide program with adjustable parameters, then we can optimize those parameters so as to minimize the distance between the joint distribution of the choices in the guide program and those in the target. For example:

```
Optimize({
  steps: 10000,
  model: function() {
    var x = sample(Gaussian({ mu: 0, sigma: 1 }), {
      guide: function() {
        return Gaussian({ mu: param(), sigma: 1 });
      }});
    factor(-(x-2)*(x-2))
    return x;
  }});
```

This general approach includes a number of well-known algorithms as special cases.

It is supported in WebPPL by *a method for performing optimization*, primitives for specifying *parameters*, and the ability to specify guides.

Optimize

Optimize(*options*)

Arguments

- **options** (*object*) – Optimization options.

Returns Nothing.

Optimizes the parameters of the guide program specified by the `model` option.

A *default guide distribution* is used for random choices that do not have a guide distribution specified explicitly.

The following options are supported:

`model`
A function of zero arguments that specifies the target and guide programs.

This option must be present.

`steps`
The number of optimization steps to take.

Default: `1`

`optMethod`
The optimization method used. The following methods are available:

- `'sgd'`
- `'adagrad'`
- `'rmsprop'`
- `'adam'`

Each method takes a `stepSize` sub-option, see below for example usage. Additional method specific options are available, see the adnn optimization module for details.

Default: `'adam'`

`estimator`
Specifies the optimization objective and the method used to estimate its gradients. See *Estimators*.

Default: `ELBO`

`weightDecay`
Specifies the strength of an L2 penalty applied to all parameters during optimization.

More specifically, a term `0.5 * strength * paramVal^2` is added to the objective for each parameter encountered during optimization. Note that this addition is not reflected in the value of the objective reported during optimization.

For parameters of the model, when the objective is the ELBO, this is equivalent to specifying a mean zero and variance `1/strength` Gaussian prior and a Delta guide for each parameter.

Default: `0`

`onStep`
Specifies a function that will be called after each step. The function will be passed the index of the current step and the value of the objective as arguments. For example:

```
var callback = function(index, value) { /* ... */ };
Optimize({model: model, steps: 100, onStep: callback});
```

If this function returns `true`, `Optimize` will return immediately, skipping any remaining optimization steps.

`verbose`
Default: `false`

Example usage:

```
Optimize({model: model, steps: 100});
Optimize({model: model, optMethod: 'adagrad'});
Optimize({model: model, optMethod: {sgd: {stepSize: 0.5}}});
```

Estimators

The following estimators are available:

ELBO

This is the evidence lower bound (ELBO). Optimizing this objective yields variational inference.

For best performance use *mapData()* in place of *map()* where possible when optimizing this objective. The conditional independence information this provides is used to reduce the variance of gradient estimates which can significantly improve performance, particularly in the presence of discrete random choices. Data sub-sampling is also supported through the use of *mapData()*.

The following options are supported:

samples

The number of samples to take for each gradient estimate.

Default: `1`

avgBaselines

Enable the "average baseline removal" variance reduction strategy.

Default: `true`

avgBaselineDecay

The decay rate used in the exponential moving average used to estimate baselines.

Default: `0.9`

Example usage:

```
Optimize({model: model, estimator: 'ELBO'});
Optimize({model: model, estimator: {ELBO: {samples: 10}}});
```

Parameters

param ([*options*])

Retrieves the value of a parameter by name. The parameter is created if it does not already exist.

The following options are supported:

dims

When `dims` is given, `param` returns a tensor of dimension `dims`. In this case `dims` should be an array.

When `dims` is omitted, `param` returns a scalar.

init

A function that computes the initial value of the parameter. The function is passed the dimension of a tensor as its only argument, and should return a tensor of that dimension.

When `init` is omitted, the parameter is initialized with a draw from the Gaussian distribution described by the `mu` and `sigma` options.

mu
: The mean of the Gaussian distribution from which the initial parameter value is drawn when `init` is omitted.

  Default: `0`

sigma
: The standard deviation of the Gaussian distribution from which the initial parameter value is drawn when `init` is omitted. Specify a standard deviation of `0` to deterministically initialize the parameter to `mu`.

  Default: `0.1`

name
: The name of the parameter to retrieve. If `name` is omitted a default name is automatically generated based on the current stack address, relative to the current coroutine.

Examples:

```
param()
param({name: 'myparam'})
param({mu: 0, sigma: 0.01, name: 'myparam'})
param({dims: [10, 10]})
param({dims: [2, 1], init: function(dims) { return ones(dims); }})
```

modelParam([*options*])
: An analog of `param` used to create or retrieve a parameter that can be used directly in the model.

  Optimizing the *ELBO* yields maximum likelihood estimation for model parameters. `modelParam` cannot be used with other inference strategies as it does not have an interpretation in the fully Bayesian setting. Attempting to do so will raise an exception.

  `modelParam` supports the same options as `param`. See the *documentation for param* for details.

Persistence

The file store provides a simple way to persist *parameters* across executions. Parameters are read from a file before the program is executed, and written back to the file once the program finishes. Enable it like so:

```
webppl model.wppl --param-store file --param-id my-parameters
```

The file used takes its name from the `param-id` command line argument (appended with `.json`) and is expected to be located in the current directory. A new file will be created if this file does not already exist.

An alternative directory can be specified using the `WEBPPL_PARAM_PATH` environment variable.

A random file name is generated when the `param-id` argument is omitted.

Parameters are also periodically written to the file during *optimization*. The frequency of writes can be controlled using the `WEBPPL_PARAM_INTERVAL` environment variable. This specifies the minimum amount of time (in milliseconds) that should elapse between writes. The default is 10 seconds.

Note that this is not intended for parallel use. The *mongo store* should be used for this instead.

Parallelization

Sharing parameters across processes

By default, parameters are stored in-memory and don't persist across executions.

As an alternative, WebPPL supports sharing parameters between WebPPL processes using MongoDB. This can be used to persist parameters across runs, speed up optimization by running multiple identical processes in parallel, and optimize multiple objectives simultaneously.

To use the MongoDB store, select it at startup time as follows:

```
webppl model.wppl --param-store mongo
```

Parameters are associated with a *parameter set id* and sharing only takes place between executions that use the same id. To control sharing, you can specify a particular id using the `param-id` command-line argument:

```
webppl model.wppl --param-store mongo --param-id my-parameter-set
```

To use the MongoDB store, MongoDB must be running. By default, WebPPL will look for MongoDB at `localhost:27017` and use the collection `parameters`. This can be changed by adjusting the environment variables `WEBPPL_MONGO_URL` and `WEBPPL_MONGO_COLLECTION`.

Running multiple identical processes in parallel

To simplify launching multiple identical processes with shared parameters, WebPPL provides a `parallelRun` script in the `scripts` folder. For example, to run ten processes that all execute `model.wppl` with parameter set id `my-parameter-set`, run:

```
scripts/parallelRun model.wppl 10 my-parameter-set
```

Any extra arguments are passed on to WebPPL, so this works:

```
scripts/parallelRun model.wppl 10 my-parameter-set --require webppl-json
```

For a few initial results on the use of parallel parameter updates for LDA, see this presentation.

CHAPTER 10

Built-in Functions

Arrays

map (*fn*, *arr*)

Returns an array obtained by mapping the function `fn` over array `arr`.

```
map(function(x) { return x + 1; }, [0, 1, 2]); // => [1, 2, 3]
```

mapData (*{data: arr[, batchSize: n]}*, *fn*)

Returns an array obtained by mapping the function `fn` over array `arr`. Each application of `fn` has an element of `arr` as its first argument and the index of that element as its second argument.

`map` and `mapData` differ in that the use of `mapData` asserts to the inference back end that all executions of `fn` are conditionally independent. This information can potentially be exploited on a per algorithm basis to improve the efficiency of inference.

`mapData` also provides an interface through which inference algorithms can support data sub-sampling. Where supported, the size of a "mini-batch" can be specified using the `batchSize` option. When using data sub-sampling the array normally returned by `mapData` is not computed in its entirety, so `undefined` is returned in its place.

Only the *ELBO* optimization objective takes advantage of `mapData` at this time.

```
mapData({data: [0, 1, 2]}, function(x) { return x + 1; }); // => [1, 2, 3]
mapData({data: data, batchSize: 10}, fn);
```

map2 (*fn*, *arr1*, *arr2*)

Returns an array obtained by mapping the function `fn` over arrays `arr1` and `arr2` concurrently. Each application of `fn` has an element of `arr1` as its first argument and the element with the same index in `arr2` as its second argument.

It is assumed that `arr1` and `arr2` are arrays of the same length. When this is not the case the behavior of `map2` is undefined.

```
var concat = function(x, y) { return x + y; };
map2(concat, ['a', 'b'], ['1', '2']); // => ['a1', 'b2']
```

mapN (*fn*, *n*)
Returns an array obtained by mapping the function `fn` over the integers `[0,1,...,n-1]`.

```
var inc = function(x) { return x + 1; };
mapN(inc, 3); // => [1, 2, 3]
```

mapIndexed (*fn*, *arr*)
Returns the array obtained by mapping the function `fn` over array `arr`. Each application of `fn` has the index of the current element as its first argument and the element itself as its second argument.

```
var pair = function(x, y) { return [x, y]; };
mapIndexed(pair, ['a', 'b']); // => [[0, 'a'], [1, 'b']]
```

reduce (*fn*, *init*, *arr*)
Reduces array `arr` to a single value by applying function `fn` to an accumulator and each value of the array. `init` is the initial value of the accumulator.

```
reduce(function(x, acc) { return x + acc; }, 0, [1, 2, 3]); // => 6
```

sum (*arr*)
Computes the sum of the elements of array `arr`.

It is assumed that each element of `arr` is a number.

```
sum([1, 2, 3, 4]) // => 10
```

product (*arr*)
Computes the product of the elements of array `arr`.

It is assumed that each element of `arr` is a number.

```
product([1, 2, 3, 4]) // => 24
```

listMean (*arr*)
Computes the mean of the elements of array `arr`.

It is assumed that `arr` is not empty, and that each element is a number.

```
listMean([1, 2, 3]); // => 2
```

listVar (*arr*[, *mean*])
Computes the variance of the elements of array `arr`.

The `mean` argument is optional. When supplied it is expected to be the mean of `arr` and is used to avoid recomputing the mean internally.

It is assumed that `arr` is not empty, and that each element is a number.

```
listVar([1, 2, 3]); // => 0.6666...
```

listStdev (*arr*[, *mean*])
Computes the standard deviation of the elements of array `arr`.

The `mean` argument is optional. When supplied it is expected to be the mean of `arr` and is used to avoid recomputing the mean internally.

It is assumed that `arr` is not empty, and that each element is a number.

```
listStdev([1, 2, 3]); // => 0.8164...
```

all (*predicate*, *arr*)
Returns `true` when all of the elements of array `arr` satisfy `predicate`, and `false` otherwise.

```
all(function(x) { return x > 1; }, [1, 2, 3]) // => false
```

any (*predicate*, *arr*)
Returns `true` when any of the elements of array `arr` satisfy `predicate`, and `false` otherwise.

```
any(function(x) { return x > 1; }, [1, 2, 3]) // => true
```

zip (*arr1*, *arr2*)
Combines two arrays into an array of pairs. Each pair is represented as an array of length two.

It is assumed that `arr1` and `arr2` are arrays of the same length. When this is not the case the behavior of `zip` is undefined.

```
zip(['a', 'b'], [1, 2]); // => [['a', 1], ['b', 2]]
```

filter (*predicate*, *arr*)
Returns a new array containing only those elements of array `arr` that satisfy `predicate`.

```
filter(function(x) { return x > 1; }, [0, 1, 2, 3]); // => [2, 3]
```

find (*predicate*, *arr*)
Returns the first element of array `arr` that satisfies `predicate`. When no such element exists `undefined` is returned.

```
find(function(x) { return x > 1; }, [0, 1, 2]); // => 2
```

remove (*element*, *arr*)
Returns a new array obtained by filtering out of array `arr` elements not equal to `element`.

```
remove(0, [0, -1, 0, 2, 1]); // => [-1, 2, 1]
```

groupBy (*eqv*, *arr*)
Splits an array into sub-arrays based on pairwise equality checks performed by the function `eqv`.

```
var sameLength = function(x, y) { return x.length === y.length; };
groupBy(sameLength, ['a', 'ab', '', 'bc']); // => [['a'], ['ab', 'bc'], ['']]
```

repeat (*n*, *fn*)
Returns an array of length `n` where each element is the result of applying `fn` to zero arguments.

```
repeat(3, function() { return true; }); // => [true, true, true]
```

sort (*arr*[, *predicate*[, *fn*]])
Returns a sorted array.

Elements are compared using < by default. This is equivalent to passing `lt` as the `predicate` argument. To sort by > pass `gt` as the `predicate` argument.

To sort based on comparisons between a function of each element, pass a function as the `fn` argument.

```
sort([3,2,4,1]); // => [1, 2, 3, 4]
sort([3,2,4,1], gt); // => [4, 3, 2, 1]

var length = function(x) { return x.length; };
sort(['a', 'ab', ''], lt, length); // => ['', 'a', 'ab']
```

sortOn (*arr*[, *fn*[, *predicate*]])

This implements the same function as `sort` but with the order of the `predicate` and `fn` parameters switched. This is convenient when you wish to specify `fn` without specifying `predicate`.

```
var length = function(x) { return x.length; };
sortOn(['a', 'ab', ''], length); // => ['', 'a', 'ab']
```

Tensors

Creation

Vector (*arr*)

Arguments

- **arr** (*array*) – array of values

Creates a tensor with dimension `[m, 1]`, where `m` is the length of `arr`.

Example:

```
Vector([1, 2, 3])
```

Matrix (*arr*)

Arguments

- **arr** (*array*) – array of arrays of values

Creates a tensor with dimension `[m, n]`, where `m` is the length of `arr` and `n` is the length of `arr[0]`.

Example:

```
Matrix([[1, 2], [3, 4]])
```

Tensor (*dims*, *arr*)

Arguments

- **dims** (*array*) – array of dimension sizes
- **arr** (*array*) – array of values

Creates a tensor with dimension `dims` out of a flat array `arr`.

Example:

```
Tensor([2, 2, 2], [1, 2, 3, 4, 5, 6, 7, 8])
```

zeros (*dims*)

Arguments

- **dims** (*array*) – dimension of tensor

Creates a tensor with dimension `dims` and all elements equal to zero.

Example:

```
zeros([10, 1])
```

ones (*dims*)

Arguments

- **dims** (*array*) – dimension of tensor

Creates a tensor with dimension `dims` and all elements equal to one.

Example:

```
ones([10, 1])
```

idMatrix (*n*)

Returns the `n` by `n` identity matrix.

oneHot (*k*, *n*)

Returns a vector of length `n` in which the `k` th entry is one and all other entries are zero.

Operations

WebPPL inherits its Tensor functionality from adnn. It supports all of the tensor functions documented here. Specifically, the `ad.tensor` module (and all the functions it contains) are globally available in WebPPL. For convenience, WebPPL also aliases `ad.tensor` to `T`, so you can write things like:

```
var x = Vector([1, 2, 3]);
var y = Vector([3, 4, 5]);
var x = T.dot(x, y);         // instead of ad.tensor.dot(x, y)
```

Other

dims (*tensor*)

Returns the shape of `tensor`.

```
dims(ones([3, 2])) // => [3, 2]
```

concat (*arr*)

Returns the vector obtained by concatenating array of vectors `arr`.

```
concat([Vector([1, 2]), Vector([3, 4])]) // => Vector([1, 2, 3, 4])
```

Neural networks

In WebPPL neural networks can be represented as simple *parameterized* functions. The language includes a number of helper functions that capture common patterns in the shape of these functions. These helpers typically take a name and the desired input and output dimensions of the network as arguments. For example:

```
var net = affine('net', {in: 3, out: 5});
var out = net(ones([3, 1])); // dims(out) == [5, 1]
```

Larger networks are built with ordinary function composition. The *stack()* helper provides a convenient way of composing multiple layers:

```
var mlp = stack([
  sigmoid,
  affine('layer2', {in: 5, out: 1}),
  tanh,
  affine('layer1', {in: 5, out: 5})
]);
```

It's important to note that the parameters of these functions are created when the constructor function (e.g. *affine()*) is called. As a consequence, models should be written such that constructors are called on every evaluation of the model. If a constructor is instead called only once before `Infer` or `Optimize` is called, then the parameters of the network will not be optimized.

```
// Correct
var model = function() {
  var net = affine('net', opts);
  /* use net */
};
Infer({model: model, /* options */});

// Incorrect
var net = affine('net', opts);
var model = function() {
  /* use net */
};
Infer({model: model, /* options */});
```

Feed forward

affine (*name, {in, out[, param, init, initb]}*)

Returns a parameterized function of a single argument that performs an affine transform of its input. This function maps a vector of length `in` to a vector of length `out`.

By default, the weight and bias parameters are created using the *param()* method. An alternative method (e.g. *modelParam()*) can be specified using the `param` option.

The `init` option can be used to specify how the weight matrix is initialized. It accepts a function that takes the shape of the matrix as its argument and returns a matrix of that shape. When the `init` option is omitted Xavier initialization is used.

The `initb` argument specifies the value with which each element of the bias vector is initialized. The default is `0`.

Example usage:

```
var init = function(dims) {
  return idMatrix(dims[0]);
};
var net = affine('net', {in: 10, out: 10, init: init, initb: -1});
var output = net(input);
```

Recurrent

These functions return a parameterized function of two arguments that maps a state vector of length `hdim` and an input vector of length `xdim` to a new state vector. Each application of this function computes a single step of a recurrent network.

rnn (*name, {hdim, xdim, [, param, output]}*)
Implements a vanilla RNN. By default the new state vector is passed through the `tanh` function before it is returned. The `output` option can be used to specify an alternative output function.

gru (*name, {hdim, xdim, [, param]}*)
Implements a gated recurrent unit. This is similar to the variant described in Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.

lstm (*name, {hdim, xdim, [, param]}*)
Implements a long short term memory. This is similar to the variant described in Generating sequences with recurrent neural networks. The difference is that here there are no peep-hole connections. i.e. The previous memory state is not passed as input to the forget, input, or output gates.

Nonlinear functions

Some nonlinear functions commonly used when building networks. Each is applied element-wise to its argument.

sigmoid (*tensor*)

tanh (*tensor*)

relu (*tensor*)

softplus (*tensor*)

softmax (*tensor*)

Other

stack (*fns*)
Returns the composition of the array of functions `fns`. The composite function applies the functions in `fns` in reverse order. That is:

```
stack([g, f]) == function(x) { return g(f(x)); }
```

Other

flip ([*p*])
Draws a sample from `Bernoulli({p: p})`.

`p` defaults to `0.5` when omitted.

uniformDraw (*arr*)
Draws a sample from the uniform distribution over elements of array `arr`.

display (*val*)
Prints a representation of the value `val` to the console.

expectation (*dist*[, *fn*])

Computes the expectation of a function `fn` under the *distribution* given by `dist`. The distribution `dist` must have finite support.

`fn` defaults to the identity function when omitted.

```
expectation(Categorical({ps: [.2, .8], vs: [0, 1]})); // => 0.8
```

marginalize (*dist*, *project*)

Marginalizes out certain variables in a distribution. `project` can be either a function or a string. Using it as a function:

```
var dist = Infer({model: function() {
  var a = flip(0.9);
  var b = flip();
  var c = flip();
  return {a: a, b: b, c: c};
}});

marginalize(dist, function(x) {
  return x.a;
}) // => Marginal with p(true) = 0.9, p(false) = 0.1
```

Using it as a string:

```
marginalize(dist, 'a') // => Marginal with p(true) = 0.9, p(false) = 0.1
```

forward (*model*)

Evaluates function of zero arguments `model`, ignoring any *conditioning*.

Also see: *Forward Sampling*

forwardGuide (*model*)

Evaluates function of zero arguments `model`, ignoring any *conditioning*, and sampling from the *guide* at each random choice.

Also see: *Forward Sampling*

mapObject (*fn*, *obj*)

Returns the object obtained by mapping the function `fn` over the values of the object `obj`. Each application of `fn` has a property name as its first argument and the corresponding value as its second argument.

```
var pair = function(x, y) { return [x, y]; };
mapObject(pair, {a: 1, b: 2}); // => {a: ['a', 1], b: ['b', 2]}
```

extend (*obj1*, *obj2*, ...)

Creates a new object and assigns own enumerable string-keyed properties of source objects 1, 2, ... to it. Source objects are applied from left to right. Subsequent sources overwrite property assignments of previous sources.

```
var x = { a: 1, b: 2 };
var y = { b: 3, c: 4 };
extend(x, y);  // => { a: 1, b: 3, c: 4 }
```

cache (*fn*, *maxSize*)

Returns a memoized version of `fn`. The memoized function is backed by a cache that is shared across all executions/possible worlds.

`cache` is provided as a means of avoiding the repeated computation of a *deterministic* function. The use of `cache` with a *stochastic* function is unlikely to be appropriate. For stochastic memoization see *mem()*.

When `maxSize` is specified the memoized function is backed by a LRU cache of size `maxSize`. The cache has unbounded size when `maxSize` is omitted.

`cache` can be used to memoize mutually recursive functions, though for technical reasons it must currently be called as `dp.cache` for this to work.

`cache` does not support caching functions of scalar/tensor arguments when performing inference with gradient based algorithms. (e.g. *HMC*, *ELBO*.) Attempting to do so will produce an error.

mem (*fn*)
Returns a memoized version of `fn`. The memoized function is backed by a cache that is local to the current execution.

Internally, the memoized function compares its arguments by first serializing them with `JSON.stringify`. This means that memoizing a higher-order function will not work as expected, as all functions serialize to the same string.

error (*msg*)
Halts execution of the program and prints `msg` to the console.

kde (*marginal*[, *kernelWidth*])
Constructs a *KDE()* distribution from a sample based marginal distribution.

CHAPTER 11

The Global Store

Background

The subset of JavaScript supported by WebPPL does not include general assignment expressions. This means it is not possible to change the value bound to a variable, or to modify the contents of a compound data structure:

```
var a = 0;
a = 1; // won't work

var b = {x: 0};
b.x = 1; // won't work
```

Attempting to do either of these things (which we will collectively refer to as 'assignment') generates an error.

This restriction isn't usually a problem as most of the things you might like to write using assignment can be expressed conveniently in a functional style.

However, assignment can occasionally be useful, and for this reason WebPPL provides a limited form of it through something called the global store.

Introducing the global store

The global store is a built-in data structure with special status in the language. It is available in all programs as `globalStore`.

Unlike regular compound data structures in WebPPL its contents *can* be modified. Here's a simple example:

```
globalStore.x = 0; // assign
globalStore.x = 1; // reassign
globalStore.x += 1;
display(globalStore.x) // prints 2
```

When reading and writing to the global store, it behaves like a plain JavaScript object. As in JavaScript, the value of each property is initially `undefined`.

Note that while the store can be modified by assigning and reassigning values to its properties, it is not possible to mutate compound data structures referenced by those properties:

```
globalStore.foo = {x: 0}
globalStore.foo = {x: 1} // reassigning foo is ok

globalStore.foo = {x: 0}
globalStore.foo.x = 1 // attempting to mutate foo fails
```

Marginal inference and the global store

Crucially, all marginal inference algorithms are aware of the global store and take care to ensure that performing inference over code that performs assignment produces correct results.

To see why this is important consider the following program:

```
var model = function() {
  var x = uniformDraw([0, 1]);
  return x;
};
```

The marginal distribution on return values for this program is:

```
Infer({method: 'enumerate'}, model);

// Marginal:
//   0 : 0.5
//   1 : 0.5
```

Now imagine re-writing this model using assignment:

```
var model = function() {
  globalStore.x = 0;
  globalStore.x += uniformDraw([0, 1]);
  return globalStore.x;
};
```

Intuitively, these programs should have the same marginal distribution, and in fact they do in WebPPL. However, the way this works is a little subtle.

To see why, let's see how inference in our simple model proceeds, keeping track of the value in the global store as we go.

For this example we will perform marginal inference by *enumeration* but something similar applies to all inference strategies.

Marginal inference by enumeration works by exploring all execution paths through the program. If the global store was shared across paths then the above example would produce counter-intuitive results.

In our example, the first path taken through the program chooses `1` from the `uniformDraw` which looks something like:

```
globalStore.x = 0;                    // {x: 0} <- state of the global store
globalStore.x += uniformDraw([0, 1]); // {x: 1} choose 1, update store
return globalStore.x;                 // Add 1 to the marginal distribution.
```

Next, we continue from the `uniformDraw` this time choosing `0`:

```
//                                  // {x: 1} carried over from previous execution
globalStore.x += uniformDraw([0, 1]) // {x: 1} choose 0, updating store produces no␣
↪change
return globalStore.x;               // Add 1 to the marginal distribution
```

All paths have now been explored, but our marginal distribution only includes `1`!

The solution is have the global store be local to each execution, so that assignment on one path is not visible from another. This is what happens in WebPPL.

Another way to think about this is to view each execution path as a possible world in a simulation. From this point of view the global store is world local; it's not possible to reach into other worlds and modify their state.

When to use the store

If you find yourself threading an argument through every function call in your program, you might consider replacing this with a value in the global store.

When not to use the global store

Maintaining a store local to each execution as described above incurs overhead.

For this reason, it is best not to use the store as a general replacement for assignment as typically used in imperative programming languages. Instead, it is usually preferable to express the program in a functional style.

Consider for example the case of concatenating an array of strings. Rather than accumulating the result in the global store:

```
var f = function() {
  var names = ['alice', 'bob'];
  globalStore.out = '';
  map(function(name) { globalStore.out += name; }, names);
  return globalStore.out;
};
```

It is *much* better to use `reduce` to achieve the same result:

```
var f = function() {
  var names = ['alice', 'bob'];
  return reduce(function(acc, name) { return acc + name; }, '', names);
};
```

CHAPTER 12

Packages

WebPPL packages are regular Node.js packages optionally extended to include WebPPL code and headers.

To make a package available in your program use the `--require` argument:

```
webppl myFile.wppl --require myPackage
```

WebPPL will search the following locations for packages:

1. The `node_modules` directory within the directory in which your program is stored.
2. The `.webppl/node_modules` directory within your home directory. Packages can be installed into this directory with `npm install --prefix ~/.webppl myPackage`.

Packages can be loaded from other locations by passing a path:

```
webppl myFile.wppl --require ../myPackage
```

Packages can extend WebPPL in several ways:

WebPPL code

You can automatically prepend WebPPL files to your code by added a `wppl` entry to `package.json`. For example:

```
{
  "name": "my-package",
  "webppl": {
    "wppl": ["myLibrary.wppl"]
  }
}
```

The use of some inference algorithms causes a caching transform to be applied to each `wppl` file. It is possible to skip the application of this transform on a per-file basis by placing the `no caching` directive at the beginning of the file. For example:

```
'no caching';

// Rest of WebPPL program
```

This is expected to be useful in only a limited number of cases and shouldn't be applied routinely.

JavaScript functions and libraries

Any regular JavaScript code within a package is made available in WebPPL as a global variable. The global variable takes the same name as the package except when the package name includes one or more - characters. In such cases the name of the global variable is obtained by converting the package name to camelCase.

For example, if the package `my-package` contains this file:

```
// index.js
module.exports = {
  myAdd: function(x, y) { return x + y; }
};
```

Then the function `myAdd` will be available in WebPPL as `myPackage.myAdd`.

If your JavaScript isn't in an `index.js` file in the root of the package, you should indicate the entry point to your package by adding a `main` entry to `package.json`. For example:

```
{
  "name": "my-package",
  "main": "src/main.js"
}
```

Note that packages must export functions as properties of an object. Exporting functions directly will not work as expected.

Additional header files

Sometimes, it is useful to define external functions that are able to access WebPPL internals. Header files have access to the following:

- The store, continuation, and address arguments that are present at any point in a WebPPL program.
- The `env` container which allows access to `env.coroutine` among other things.

Let's use the example of a function that makes the current address available in WebPPL:

1. Write a JavaScript file that exports a function. The function will be called with the `env` container and should return an object containing the functions you want to use:

```
// addressHeader.js

module.exports = function(env) {

  function myGetAddress(store, k, address) {
    return k(store, address);
  };

  return { myGetAddress: myGetAddress };
```

```
};
```

2. Add a `headers` entry to `package.json`:

```
{
  "name": "my-package",
  "webppl": {
    "headers": ["addressHeader.js"]
  }
}
```

3. Write a WebPPL file that uses your new functions (without module qualifier):

```
// addressTest.wppl

var foo = function() {
  var bar = function() {
    console.log(myGetAddress());
  }
  bar();
};

foo();
```

Package template

The WebPPL package template provides a scaffold that you can extend to create your own packages.

Useful packages

- json: read/write json files
- csv: read/write csv files
- fs: read/write files in general
- dp: dynamic programming (caching for mutually recursive functions)
- editor: browser based editor
- viz: visualization utilities
- bda: data analysis utilities
- agents: agent simulations
- timeit: timing utilities
- intercache: interpolating cache
- oed: optimal experimental design

These packages are no longer maintained, but may be worth a look:

- caches: cache inference results to disk
- formal: static analysis in Racket for WebPPL

- isosmc: utils for defining sequences of distributions for smc

CHAPTER 13

Workflow

Installation from GitHub

```
git clone https://github.com/probmods/webppl.git
cd webppl
npm install
npm install -g nodeunit grunt-cli
```

To use the `webppl` command line tool from any directory, add the webppl directory to your `$PATH`.

Updating the npm package

1. Get latest dev version:

   ```
   git checkout dev
   git pull
   ```

2. Merge into master and test:

   ```
   git checkout master
   git pull
   git merge dev
   grunt
   ```

3. Update version number on master:

   ```
   npm version patch  # or minor, or major; prints new version number
   ```

4. Merge updated version number into dev:

   ```
   git checkout dev
   git merge master
   ```

4. Push to remotes and npm:

```
git push origin dev
git push origin master
git push origin v0.0.1  # use version printed by "npm version" command above
npm publish
```

Committing changes

Before committing changes, run grunt (which runs *tests* and *linting*):

```
grunt
```

If grunt doesn't succeed, the continuous integration tests will fail as well.

Modifying .ad.js files

Files with names which end with `.ad.js` are transformed to use AD primitives when WebPPL is installed.

During development it is necessary to run this transform after any such files have been modified. A grunt task is provided that will monitor the file system and run the transform when any `.ad.js` files are updated. Start the task with:

```
grunt build-watch
```

Alternatively, the transform can be run directly with:

```
grunt build
```

The scope of the transform is controlled with the `'use ad'` directive. If this directive appears directly after the `'use strict'` directive at the top of a file, then the whole file will be transformed. Otherwise, those functions which include the directive before any other statements or expressions in their body will be transformed. Any function nested within a function which includes the directive will also be transformed.

Tests

To only run the tests, do:

```
npm test
```

To reproduce intermittent test failures run the inference tests with the random seed displayed in the test output. For example:

```
RANDOM_SEED=2344512342 nodeunit tests/test-inference.js
```

nodeunit can also run individual tests or test groups. For example:

```
nodeunit tests/test-inference.js -t Enumerate
```

See the nodeunit documentation for details.

Linting

To only run the linter:

```
grunt lint
```

For more semantic linting, try:

```
grunt hint
```

If the linter complains about style errors (like indentation), you can fix many of them automatically using:

```
grunt lint --fix --force
```

Browser version

To generate a version of WebPPL for in-browser use, run:

```
npm install -g browserify uglify-js
grunt bundle
```

The output is written to `bundle/webppl.js` and a minified version is written to `bundle/webppl.min.js`.

To use in web pages:

```
<script src="webppl.js"></script>
<script>webppl.run(...)</script>
```

We also provide an in-browser editor for WebPPL code.

Testing

To check that compilation was successful, run the browser tests using:

```
grunt test-browser
```

The tests will run in the default browser. Specify a different browser using the `BROWSER` environment variable. For example:

```
BROWSER="Google Chrome" grunt test-browser
```

Incremental compilation

Repeatedly making changes to the code and then testing the changes in the browser can be a slow process. watchify speeds up this process by performing an incremental compile whenever it detects changes to source files. To start watchify use:

```
npm install -g watchify
grunt browserify-watch
```

Note that this task only updates `bundle/webppl.js`. Before running the browser tests and deploying, create the minified version like so:

```
grunt uglify
```

Packages

Packages can also be used in the browser. For example, to include the `webppl-viz` package use:

```
grunt bundle:path/to/webppl-viz
```

Multiple packages can specified, separated by colons.

Bibliography

[wingate11] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. "Lightweight implementations of probabilistic programming languages via transformational compilation." International Conference on Artificial Intelligence and Statistics. 2011.

[neal11] Radford M. Neal, "MCMC using Hamiltonian dynamics." Handbook of Markov Chain Monte Carlo 2 (2011).

[ritchie15] Daniel Ritchie, Andreas Stuhlmüller, and Noah D. Goodman. "C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching." International Conference on Artificial Intelligence and Statistics. 2016.

Index

O

P

Q

R

S

T

U

V

Z