Contents

1 Getting Started 1
  1.1 Try WebPPL ......................................................... 1
  1.2 Learning .......................................................... 1
  1.3 Need help? .......................................................... 1

2 Language Overview 3
  2.1 Syntax ............................................................ 3
  2.2 Calling JavaScript Functions ..................................... 4

3 Installation 5
  3.1 Updating .......................................................... 5

4 Usage 7
  4.1 Arguments .......................................................... 7
  4.2 Passing arguments to the program ............................... 7

5 Debugging 9

6 Sample 11
  6.1 Guides ............................................................ 11
  6.2 Drift Kernels ....................................................... 12

7 Distributions 15
  7.1 Primitives .......................................................... 15

8 Inference 21
  8.1 Methods ............................................................ 21
  8.2 Conditioning ....................................................... 27

9 Optimization 29
  9.1 Optimize ........................................................... 29
  9.2 Parameters ........................................................ 31
  9.3 Persistence ........................................................ 32
  9.4 Parallelization ..................................................... 33

10 Built-in Functions 35
  10.1 Arrays ............................................................ 35
  10.2 Tensors ............................................................ 38
1.1 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.

1.2 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.

1.3 Need help?

If you have any questions about installing WebPPL or need help with your code, you can get help on the Google group.
The WebPPL language begins with a subset of JavaScript, and adds to it primitive distributions and operations to perform sampling, conditioning and inference.

2.1 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:

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

```javascript
var foo = function(x) {
    var bar = Math.exp(x);
    var baz = x === 0 ? [] : [Math.log(bar), foo(x-1)];
    return baz;
}
foo(5);
```

## 2.2 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')`. 
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 (node package manager) to install WebPPL:

```
npm install -g webppl
```

Create a file called `test.wppl`:

```javascript
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
```

### 3.1 Updating

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

```
npm update -g webppl
```
Run WebPPL programs:

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

## 4.1 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
```

## 4.2 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:
When run with:

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

Will produce the following output:

```json
{ 
  _: ['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.
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.
6.1 Guides

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

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

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

For example:

```plaintext
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:

```plaintext
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.
### 6.1.1 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:

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

### 6.2 Drift Kernels

#### 6.2.1 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 is 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.

#### 6.2.2 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:

```javascript
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:

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

To use our `gaussianKernel` with a Cauchy random choice we would write:
6.2.3 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.

\[
\text{gaussianDrift} \left( \{ \mu: \ldots, \sigma: \ldots, \text{width:} \ldots \} \right)
\]

\[
\text{dirichletDrift} \left( \{ \alpha: \ldots, \text{concentration:} \ldots \} \right)
\]

\[
\text{uniformDrift} \left( \{ a: \ldots, b: \ldots, \text{width:} \ldots \} \right)
\]

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:

```plaintext
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:

```plaintext
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:

```plaintext
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.
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:

```javascript
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.

### 7.1 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 \( \text{Bernoulli}(p: p) \) trials.

**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.**

**Distribution over \([-\infty, \infty]\)**

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

**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.**

**Distribution over probability vectors. If \( \alpha \) has length \( d \) then the distribution is over probability vectors of length \( d \).**

**Distribution over \([0, \infty]\)**

**A rate (real (0, Infinity))**

**Chapter 7. Distributions**
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** \(\{\text{dists: ...}, \text{ps: ...}\}\)
- \text{dists}: array of component distributions
- \text{ps}: component probabilities (can be unnormalized) \((\text{vector or real array} \ [0, \text{Infinity})\))

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** \(\{\text{ps: ...}, \text{n: ...}\}\)
- \text{ps}: probabilities \((\text{real array with elements that sum to one})\)
- \text{n}: number of trials \((\text{int} \ (\geq 1))\)

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

Wikipedia entry

**MultivariateBernoulli** \(\{\text{ps: ...}\}\)
- \text{ps}: probabilities \((\text{vector} \ [0, 1])\)

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 \text{Bernoulli} which has support \{true, false\}.

**MultivariateGaussian** \(\{\mu: \ldots, \text{cov: ...}\}\)
- \text{mu}: mean \((\text{vector})\)
- \text{cov}: covariance \((\text{positive definite matrix})\)

Multivariate Gaussian distribution with full covariance matrix. If \(\mu\) has length \(d\) and \(\text{cov}\) is a \(d\)-by-\(d\) matrix, then the distribution is over vectors of length \(d\).

Wikipedia entry

**Poisson** \(\{\mu: \ldots\}\)
- \text{mu}: mean \((\text{real} \ (0, \text{Infinity}))\)

Distribution over integers.

Wikipedia entry

**RandomInteger** \(\{n: \ldots\}\)
- \text{n}: number of possible values \((\text{int} \ (\geq 1))\)

Uniform distribution over \(\{0, 1, \ldots, n-1\}\)

Wikipedia entry

**TensorGaussian** \(\{\mu: \ldots, \text{sigma: ...}, \text{dims: ...}\}\)
- \text{mu}: mean \((\text{real})\)
- \text{sigma}: standard deviation \((\text{real} \ (0, \text{Infinity}))\)
- \text{dims}: dimension of tensor \((\text{int} \ (\geq 1) \ array)\)

Distribution over a tensor of independent Gaussian variables.

**TensorLaplace** \(\{\text{location: ...}, \text{scale: ...}, \text{dims: ...}\}\)
- \text{location}: \((\text{real})\)
- \text{scale}: \((\text{real} \ (0, \text{Infinity}))\)
- \text{dims}: dimension of tensor \((\text{int} \ (\geq 1) \ array)\)
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
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:

```javascript
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:

### 8.1 Methods

#### 8.1.1 Enumeration

```javascript
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});
```

8.1.2 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});
```

8.1.3 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:

```javascript
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:

```javascript
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:

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

Infer({method: 'MCMC', kernel: {HMC: {steps: 10, stepSize: 1}}, model: model});
```
8.1.4 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

### 8.1.5 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:**

```javascript
Infer({method: 'SMC', particles: 100, rejuvSteps: 5, model: model});
```
8.1.6 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});
```

8.1.7 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});
```
8.2 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:

```javascript
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:

```javascript
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:

```javascript
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.
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:

```javascript
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.

### 9.1 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 $0.5 \times strength \times \text{paramVal}^2$ 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 $1/\text{strength}$ 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:

```javascript
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}}});

9.1.1 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:

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

9.2 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 \texttt{init} is omitted.
Default: 0

sigma
The standard deviation of the Gaussian distribution from which the initial parameter value is drawn when \texttt{init} is omitted. Specify a standard deviation of 0 to deterministically initialize the parameter to \texttt{mu}.
Default: 0.1

name
The name of the parameter to retrieve. If \texttt{name} is omitted a default name is automatically generated based on the current stack address, relative to the current coroutine.

Examples:

\begin{verbatim}
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); } })
\end{verbatim}

\begin{verbatim}
modelParam([options])
An analog of \texttt{param} used to create or retrieve a parameter that can be used directly in the model.

Optimizing the \textit{ELBO} yields maximum likelihood estimation for model parameters. \texttt{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.

\texttt{modelParam} supports the same options as \texttt{param}. See the documentation for \texttt{param} for details.
\end{verbatim}

9.3 Persistence

The file store provides a simple way to persist \textit{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:

\begin{verbatim}
webppl model.wppl --param-store file --param-id my-parameters
\end{verbatim}

The file used takes its name from the \texttt{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 \texttt{WEBPPL_PARAM_PATH} environment variable.

A random file name is generated when the \texttt{param-id} argument is omitted.

Parameters are also periodically written to the file during \textit{optimization}. The frequency of writes can be controlled using the \texttt{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 \textit{mongo store} should be used for this instead.
9.4 Parallelization

9.4.1 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.

9.4.2 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.
10.1 Arrays

**map** *(fn, arr)*

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

```javascript
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.

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

**mapN (fn, n)**

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

```javascript
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.

```javascript
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.

```javascript
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.

```javascript
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.

```javascript
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.

```javascript
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.

```javascript
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.

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

**all** *(predicate, arr)*

Returns \texttt{true} when all of the elements of array \( arr \) satisfy \( predicate \), and \texttt{false} otherwise.

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

**any** *(predicate, arr)*

Returns \texttt{true} when any of the elements of array \( arr \) satisfy \( predicate \), and \texttt{false} otherwise.

```javascript
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.

```javascript
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 \( predicate \).

```javascript
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 \texttt{undefined} is returned.

```javascript
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 \( element \).

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

**groupBy** *(equiv, arr)*

Splits an array into sub-arrays based on pairwise equality checks performed by the function \( equiv \).

```javascript
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.

```javascript
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 \( \texttt{lt} \) as the \( predicate \) argument. To sort by \( >\) pass \( \texttt{gt} \) as the \( predicate \) argument.

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

10.1. Arrays 37
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']

10.2 Tensors

10.2.1 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** `

Arguments

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

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

Example:

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

**idMatrix** `

Returns the \(n \times n\) identity matrix.

**oneHot** `

Returns a vector of length \(n\) in which the \(k^{\text{th}}\) entry is one and all other entries are zero.

### 10.2.2 Operations

WebPPL inherits its Tensor functionality from `ad.tensor`. 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 = T.transpose(Vector([1, 2, 3])); // instead of ad.tensor.transpose
var y = Vector([3, 4, 5]);
T.dot(x, y); // instead of ad.tensor.dot
```

### 10.2.3 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])
```

### 10.3 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:

```javascript
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.

```javascript
// 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 */});
```

### 10.3.1 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:

```javascript
var init = function(dims) {
  return idMatrix(dims[0]);
};
var net = affine('net', {in: 10, out: 10, init: init, initb: -1});
var output = net(input);
```
10.3.2 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.

10.3.3 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)*

10.3.4 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)); }
```

10.4 Other

**flip** *(p)*

Draws a sample from $\text{Bernoulli}(p)$.

$p$ 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.

```webpl
evaluation(Categorical({ps: [0.2, 0.8], 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:

```webpl
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:

```webpl
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.

```webpl
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.

```webpl
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.

**AIS** `(model[, options])`  
Returns an estimate of the log of the normalization constant of `model`. This is not an unbiased estimator, rather it is a stochastic lower bound. [grosse16]

The sequence of intermediate distributions used by AIS is obtained by scaling the contribution to the overall score made by the `factor` statements in `model`.

When a model includes hard factors (e.g. `factor(-Infinity), condition(bool)`) this approach does not produce an estimate of the expected quantity. Hence, to avoid confusion, an error is generated by AIS if a hard factor is encountered in the model.

The length of the sequence of distributions is given by the `steps` option. At step `k` the score given by each `factor` is scaled by `k / steps`.

The MCMC transition operator used is based on the MH kernel.

The following options are supported:

**steps**  
The length of the sequence of intermediate distributions.  
Default: 20

**samples**  
The number of times the AIS procedure is repeated. AIS returns the average of the log of the estimates produced by the individual runs.  
Default: 1

Example usage:

```
AIS(model, {samples: 100, steps: 100})
```

**Bibliography**

10.4. Other 43
11.1 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:

```javascript
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.

11.2 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:

```javascript
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
```

### 11.3 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:

```javascript
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:

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

```plaintext
// {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.

### 11.4 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.

### 11.5 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:

```plaintext
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:

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

### 12.1 WebPPL code

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

```
{
  "name": "my-package",
  "wppl": {
    "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:
This is expected to be useful in only a limited number of cases and shouldn’t be applied routinely.

## 12.2 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:

```javascript
// 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:

```json
{
    "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.

### 12.3 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:

```javascript
// addressHeader.js
module.exports = function(env) {
    function myGetAddress(store, k, address) {
        return k(store, address);
    }
};
```

(continues on next page)
(continued from previous page)

```javascript
return { myGetAddress: myGetAddress }
};
```

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

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

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

```javascript
// addressTest.wppl
var foo = function() {
  var bar = function() {
    console.log(myGetAddress());
  }
  bar();
};
foo();
```

### 12.4 Package template

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

#### 12.5 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

---

12.4. Package template 51
• **formal**: static analysis in Racket for WebPPL
• **isosmc**: utils for defining sequences of distributions for smc
CHAPTER 13

Workflow

13.1 Installation from GitHub

```
git clone https://github.com/probmods/webppl.git
cd webppl
npm --version # ensure >= 4.0.0 (required to run build scripts)
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.

13.2 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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>git push origin dev</td>
<td>Push to origin dev</td>
</tr>
<tr>
<td>git push origin master</td>
<td>Push to origin master</td>
</tr>
<tr>
<td>git push origin v0.0.1</td>
<td>Push version v0.0.1 with the specified version</td>
</tr>
<tr>
<td>npm --version</td>
<td>Ensure npm version is greater than or equal to</td>
</tr>
<tr>
<td></td>
<td>4.0.0 (required to run build scripts)</td>
</tr>
<tr>
<td>npm publish --unsafe-perm</td>
<td>Prevent scripts from failing when npm is run as</td>
</tr>
<tr>
<td></td>
<td>root</td>
</tr>
</tbody>
</table>

## 13.3 Committing changes

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

```bash
grunt
```

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

## 13.4 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:

```bash
grunt build-watch
```

Alternatively, the transform can be run directly with:

```bash
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.

## 13.5 Tests

To only run the tests, do:

```bash
npm test
```

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

```bash
RANDOM_SEED=234512342 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.

13.6 Linting

To only run the linter:

```bash
grun lint
```

For more semantic linting, try:

```bash
grun hint
```

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

```bash
grun lint --fix --force
```

13.7 Browser version

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

```bash
npm install -g browserify uglify-js
grun 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:

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

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

13.7.1 Testing

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

```bash
grun test-browser
```

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

```bash
BROWSER="Google Chrome" grun test-browser
```

13.7.2 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

### 13.7.3 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.


A
affine() (built-in function), 40
AIS() (built-in function), 43
all() (built-in function), 37
any() (built-in function), 37
B
Bernoulli() (built-in function), 15
Beta() (built-in function), 15
Binomial() (built-in function), 15
C
cache() (built-in function), 42
Categorical() (built-in function), 16
Cauchy() (built-in function), 16
concat() (built-in function), 39
condition() (built-in function), 27
D
Delta() (built-in function), 16
DiagCovGaussian() (built-in function), 16
dims() (built-in function), 39
Dirichlet() (built-in function), 16
dirichletDrift() (built-in function), 13
Discrete() (built-in function), 16
display() (built-in function), 41
E
error() (built-in function), 43
expectation() (built-in function), 41
Exponential() (built-in function), 16
extend() (built-in function), 42
F
factor() (built-in function), 27
filter() (built-in function), 37
find() (built-in function), 37
flip() (built-in function), 41
forward() (built-in function), 42
forwardGuide() (built-in function), 42
G
Gamma() (built-in function), 16
Gaussian() (built-in function), 17
gaussianDrift() (built-in function), 13
groupBy() (built-in function), 37
gpu() (built-in function), 41
I
idMatrix() (built-in function), 39
Infer() (built-in function), 21, 22, 24–26
K
KDE() (built-in function), 17
dex() (built-in function), 43
L
Laplace() (built-in function), 17
listMean() (built-in function), 36
listStdev() (built-in function), 36
listVar() (built-in function), 36
LogisticNormal() (built-in function), 17
LogitNormal() (built-in function), 17
lstm() (built-in function), 41
M
map() (built-in function), 35
map2() (built-in function), 35
mapData() (built-in function), 35
mapIndexed() (built-in function), 36
mapN() (built-in function), 36
mapObject() (built-in function), 42
marginalize() (built-in function), 42
Matrix() (built-in function), 38
mem() (built-in function), 43
Mixture() (built-in function), 17
modelParam() (built-in function), 32
Multinomial() (built-in function), 18
MultivariateBernoulli() (built-in function), 18  
MultivariateGaussian() (built-in function), 18

O
observe() (built-in function), 27
oneHot() (built-in function), 39
ones() (built-in function), 39
Optimize() (built-in function), 29

P
param() (built-in function), 31
Poisson() (built-in function), 18
product() (built-in function), 36

Q
query.add() (query method), 24

R
RandomInteger() (built-in function), 18
reduce() (built-in function), 36
relu() (built-in function), 41
remove() (built-in function), 37
repeat() (built-in function), 37
rnn() (built-in function), 41

S
sigmoid() (built-in function), 41
softmax() (built-in function), 41
softplus() (built-in function), 41
sort() (built-in function), 37
sortOn() (built-in function), 38
stack() (built-in function), 41
sum() (built-in function), 36

T
tanh() (built-in function), 41
Tensor() (built-in function), 38
TensorGaussian() (built-in function), 18
TensorLaplace() (built-in function), 18

U
Uniform() (built-in function), 19
uniformDraw() (built-in function), 41
uniformDrift() (built-in function), 13

V
Vector() (built-in function), 38

Z
zeros() (built-in function), 38
zip() (built-in function), 37