Artifact for Multi-Language Probabilistic Programming

Many different probabilistic programming languages exist that specialize to specific kinds of probabilistic programs, broadly falling into the categories of approximate and exact inference.

This artifact for Multi-Language Probabilistic Programming provides the MultiPPL compiler. MultiPPL is a host compiler of two syntactically and semantically different probabilistic programming languages: an approximate language leveraging importance sampling, and an exact language using binary decision diagrams (BDDs) for knowledge compilation. Our work demonstrates sound interoperation of these two languages under a Matthews and Findler-style multi-language framework[3].

Contents

1 Artifact Availability

The provided artifact contains the following source, development dependencies, and executables:

- This README
- The multippl source code as a separate `multippl-source.tar.gz` file.
- A docker image containing the following
  - executables for development: `cargo`, `rustc`, `tree-sitter`, `cargo-nextest`, `ghc`, `bc`
  - the multippl source code, located at `/data/multippl-source`
  - executables for benchmarking:
    * `multippl` our software artifact
    * `python` with `pyro` preinstalled, our benchmark's approximate inference alternative
    * `psi`, our benchmark's exact inference alternative
    * `dice`, used to derive components of the ground truth.
    * `multippl-benchmark`, a shell script which runs the benchmarks and tabulates our results.

2 Quick Start

2.1 Hardware Requirements

There are no explicit hardware requirements for the `mulippl` compiler. Large exact inference programs will eventually encounter memory limitations and slow down the samples produced, but this has not been an issue for networks in our evaluations.

2.2 Running `multippl`

The attached docker image has it's entrypoint set to the `multippl` binary:

```
$ wget https://zenodo.org/records/<TBD>/files/multippl-docker.tar.gz
$ docker load --input multippl-docker.tar.gz
$ docker run --rm multippl:latest --steps <STEPS> --file <FILE>
```

Will invokes `multippl`, where `--step` is the number of samples taken in the evaluation, and `--file` refers to a valid MultiPPL program.

To run `multippl` on a file outside of the docker image you must bind a docker mount to the image. Given a file `example/beta-bernoulli.yo`, bind the mount using `-v` or `--volume <HOSTPATH>:<BINDPATH>`:

```
$ docker run --rm -v $PWD/examples:/data/examples multippl:latest \
             --file /data/examples/beta-bernoulli.yo --steps 1000
0.33817887009669956
37ms
```

Notes:

- `--rm` removes all state from the docker container after execution
- `--file` can be relative to the `/data` directory.
- `/data` contains `multippl-source/` so binding directly to `/data` will lose access to these provided source files.

2.3 Running `multippl-benchmark`

To run a our benchmarks single-threaded for 100 runs and cache the resulting tables in a local `logs/` directory, use the following command:

```
$ docker run -v $PWD/logs:/data/logs --entrypoint multippl-benchmark \
             multippl:latest all --logdir /data/logs
```

PSI will time out on 700 evaluations, which is the most time consuming portion of the benchmark. To reduce the running time, it may be prudent to limit PSI to only 10 runs, relying on the provided standard error to reflect the numbers provided:

```
$ docker run -v $PWD/logs:/data/logs --entrypoint multippl-benchmark \
             multippl:latest all --logdir /data/logs --psi-runs 10
```

To do a "quick evaluation" with only 10 runs, a separate flag exists for the rest of the benchmark:

```
$ docker run -v $PWD/logs:/data/logs --entrypoint multippl-benchmark \
            multippl:latest all --logdir /data/logs --num-runs 10 --psi-runs 10
```

Tables will be cached to `logs/hybrid.rich` and `logs/discrete.rich`.

3 MultiPPL

MultiPPL uses `tree-sitter` to parse syntax with the full grammar defined in `tree-sitter-multippl/grammar.js`. Here we describe each sub-language in our framework and how to interoperate between the Cont and Disc languages by example, and provide a summarization of the tree-sitter grammar provided. The `examples/` folder contains all programs documented.

—

A MultiPPL program is introduced using `exact { ... }` or `sample { ... }` blocks:

```
choice('exact, 'sample') '{' <expr> '}'
```

Here, `choice` denotes an alternative and comes from the tree-sitter metalanguage, while `<expr>` comes from the chosen expression language of Disc (`exact`) or Cont (`sample`). Single-quoted characters denote requisite symbols.

MultiPPL supports procedures, which similarly require a `sample` or `exact` keyword to describe where the function is allowed to run:

```
choice('sample', 'exact') 'fn' <id> '(' repeat(<id>) ')' '{' <expr> '}'
```

From tree-sitter we use `repeat` for zero-or-more repetitions, `<id>` is a placeholder for a variable (defined as `x` in each grammar), and `<expr>` once again corresponds to the chosen expression language.

3.1 The Disc Language

Disc syntax draws heavily from the Dice programming language [2] and Disc's inference strategy uses the same knowledge compilation engine [1] used by Dice. The largest difference between the two languages Disc is currently untyped (the type-system will arrive in the next release). Disc allows for dynamically-allocated floating point values obtained by interoperation but querying these result in undefined behavior. The syntactic differences between Disc and Dice include:

- `observe` statements are not bound expressions, but are instead statements.
- integers are not bit-encoded and do not need an explicit size.

3.1.1 Example: Two Coins

An illustrative, simple Disc program will flip two biased coins and observe an event that one of the two coins will land on heads:

```
exact {
  let a = flip 1.0 / 3.0 in
  let b = flip 1.0 / 4.0 in
```

```
  observe a || b in
  a
}
```

In this program `flip` will represent a coinflip with the probability of heads being `flip`'s parameter; the first line of the program will create a Bernoulli distribution which returns true (ie: "heads") with probability 1/3 and bind this to `a`; on the next line we similarly create a Bernoulli distribution that is true with probability 1/4 and assign this to `b`. Next, `observe` encodes evidence that one of these variables *must* be true and the program queries for the posterior of `a`'s distribution.

We can analytically derive the solution (or construct a probability table) to show that the posterior of this model is 2/3. Running this program with the MultiPPL compiler, we see:

```
$ docker run multippl:latest --file examples/two-coins.yo --steps 1
0.6666666666666666
3ms
```

In contrast to the Dice compiler, MultiPPL will take this program and produce a sampler which executes the program for as many samples as indicated by the `--steps` flag. For this reason, the command above only returns the expectation of the compiled distribution and does not return a representation of the underlying probability table. The final line reports the wall-clock time of execution.

Because we are compiling a Disc program, the sampled distribution is exact and is invariant to the requested number of samples. If we increase the expected number of samples to 10, we will observe that this is the same as compiling the exact distribution 10 times and taking the average of these (identical) samples.

```
$ docker run multippl:latest --file examples/two-coins.yo --steps 10
0.6666666666666667
4ms
```

3.1.2 Products

Disc supports products and projections, and we can use this to query for `b`'s expectation as well:

```
exact {
  let a = flip 1.0 / 3.0 in
  let b = flip 1.0 / 4.0 in
  let ab = (a, b) in
  observe a || b in
  (ab[0], ab[1])
}
```

Compiling this query, will yield a space-delineated list of results and inform us that `b`'s posterior mean is 0.5:

```
$ docker run multippl:latest --file examples/two-coins-prod.yo --steps 1
0.6666666666666666 0.5
5ms
```

3.1.3 The Discrete Distribution and Probabilistic Choice

The Discrete distribution takes in a list of floats, normalizes this list so that they form a valid probability distribution, and returns an integer. Integers in Disc, however, are syntactic sugar for one-hot encodings of the represented int. For example the following program:

```
exact {
  discrete(1.5, 1.5, 3.0)
}
```

Is a valid query:

```
$ docker run multippl:latest --file examples/discrete.yo --steps 1
0.25 0.25 0.5
6ms
```

If-then-else expressions in Disc denote probabilistic choice.

```
exact {
  let p = flip 0.5 in
  if p
  then discrete(1.5, 1.5, 3.0)
  else discrete(3.0, 1.5, 1.5)
}
```

Probabilistic choice introduces some nuance and a longer discussion of probabilistic choice in the context of our core grammar can be found in our OOPSLA submission.

3.1.4 Grammar

A top-level summarization of Disc's grammar is as follows:

```
Variables x

Expressions
e := a                                      // all ANF forms
  | x '()' | x '(' repeat(a ',') a ')'      // function application
  | 'if' a 'then' e 'else' e                // choice
```

```
  | 'let' x '=' e 'in' e                        // variable binding
  | 'flip' a                                    // Bernoulli distributions
 | 'discrete' '(' repeat(a ',') a ')'      // Discrete distributions, desugared into a sequence of
 | 'observe' a 'in' e                 // conditioning on hard evidence in a sequence
 | 'sample' '(' sample_e ')'            // inlined interoperation with an expression e from Cont
 | 'sample' '{' sample_e '}'            // interoperation with a block expression e from Cont


ANF forms
a := x                                  // variables
  | v                                   // values
  | '!' a                               // negation
  | '(' repeat(a ',') a ')'             // products
  |  x '[' a ']'                        // projections out of products
  | a binop a                           // binomial operations

Binomial operations
binop := '+' | '*' | '/' | '^' | '<' | '<=' | '==' | '>=' | '>' | '&&' | '||'

Values
v := true | false                       // booleans
 | /-?\d+\.(?:\d*|)/           // statically known floating-point values, or floats obtained throu
 | /\d+/                    // statically known integers, or integers obtained through interop
  | '()' | '(' repeat(v ',') v ')'   // products
```

3.2 The Cont Language

The Cont language is a simple sampling language that uses importance sampling as its approximate inference strategy. It contains common distributions-objects, both continuous and discrete, as well as the ability to incorporate soft-evidence, sample from distributions, while-loops, and conventional branching statements.

3.2.1 An Approximate Beta-Bernoulli

A example of using Cont to find the posterior of a Beta-Bernoulli process, would like the following:

```
sample {
  p <- ~ beta(1.0, 1.0);
  observe true  from bern(p);
  observe false from bern(p);
  observe false from bern(p);
  p
}
```

This program first samples from a Beta(1, 1) distribution with the unary `~` operator. The result is a value with uniform probability between 0 and 1, which is assigned to the variable `p`, which will be used to parameterize the Bernoulli's distribution. This distribution then incorporates three observations into its importance weighting, which is used to score the final query's posterior which is the final line of the program. Because of conjugacy, we know that the correct posterior is a Beta distribution with

$$\alpha = 2$$

and

$$\beta = 3$$

, with an expectation of

$$2/(2+3) = 0.4$$

.

1. running Running `multippl`, we see that 100 samples produces the following expectation of the posterior:

   ```
   $ docker run multippl:latest --rng 1 \
                --file examples/beta-bernoulli.yo --steps 100
   0.3899433561293662
   7ms
   ```

   In this command, `--rng 1` indicates a seed, `--file` points to the relative path of the program in the docker container, and `--steps 100` defines the number of samples to produce. Increasing this number of samples, we see that our approximation converges closer to the correct value:

   ```
   $ docker run multippl:latest --rng 1 \
            --file examples/beta-bernoulli.yo --steps 10000
   0.3989326008738859
   535ms
   ```

3.2.2 While-loops

Four data points for inference is quite limited, requiring many samples to produce an adequate result. We may want to increase how much evidence we give our program with Cont's while-loop:

```
sample {
  p ~ beta(1.0, 1.0);
  x <- 10;
  while (x > 0) {
```

```
    observe true  from bern(p);
    observe false from bern(p);
    observe false from bern(p);
    x <- x - 1;
    ()
  };
  p
}
```

In the first line of our program, we use a binding `~` which is syntactic sugar for `p <- ~ beta(1.0, 1.0)`. Notably, all Cont statements terminate with semicolons including `while`-loops – this differs from conventional imperative programs. All blocks also return expressions and so here we provide unit `()` to the block in this while-loop, which always discards it's final value. The posterior of this program is Beta(1+10, 1+20) with a mean of

$$11/32 = 0.34375$$

```
docker run multippl:latest --rng 1 \
       --file examples/beta-bernoulli-loop.yo --steps 10000
0.34227573553622553
732ms
```

3.2.3 Branching and Lists

Cont supports branching and control flow through if statements. To define a multi-modal Gaussian distribution, we can use samples from a Bernoulli distribution, and use this to select one of two modes:

```
sample {
  m ~ bern(0.5);
  if m {
    ~normal(1.0, 0.5)
  } else {
    ~normal(-1.0, 0.5)
  }
}
```

To perform parameter estimation for this model, we would want to write some function to perform the same scoring over both modes:

```
sample fn score (p, ev) {
  m ~ bern(p);
  if m {
    observe ev from normal(1.0, 0.5); ()
  } else {
```

```
    observe ev from normal(-1.0, 0.5); ()
  }
}
sample {
  p ~ beta(1.0, 1.0);
  score(p, 1.0);
  score(p, 1.0);
  score(p, 1.0);
  p
}
```

The three observations above will begin to skew our posterior towards the Gaussian distribution with a mode of 1.0:

```
docker run multippl:latest --rng 1 \
       --file examples/multimodal.yo --steps 1000
0.8051300094638457
56ms
```

Cont has limited support for lists and includes the `head`, `tail`, and `push` functions. We can represent the same program above with a list of our evidence and iterate through this list using a `while` loop:

```
sample fn score (p, ev) {
  m ~ bern(p);
  if m {
    observe ev from normal(1.0, 0.5); ()
  } else {
    observe ev from normal(-1.0, 0.5); ()
  }
}
sample {
  p ~ beta(1.0, 1.0);
  evidence <- [1.0, 1.0, 1.0];
  i <- 3;
  while (i > 0) {
    score(p, evidence[i - 1]);
    i <- i - 1;
    ()
  };
  p
}
```

And we can confirm that running this program with the same seed will yield the same result as before:

```
docker run multippl:latest --rng 1 \
       --file examples/multimodal-iter.yo --steps 1000
0.8051300094638457
79ms
```

3.2.4 Grammar

A simplified summary of Cont's tree-sitter grammar is as follows:

```
Variables x

Expressions
e := a                                          // all ANF forms
  | 'while' a '{' e '}'                         // while loops
  | x '()' | x '(' repeat(x ',') x ')'          // function application
  |'if' '(' a ')' '{' e '}' 'else' '{' e '}'    // control flow
  | x '<-' e ';' e                              // variable binding
  | e ';' e                                     // sequencing
  | '~' e                                       // sampling an expression
  | x '~' e ';' e                       // sugar for binding a sample: x <- (~ e); e
  | 'observe' a 'from' a                        // conditioning on soft evidence
  | 'exact' '(' exact_e ')'             // inlined interoperation with an expression e from Disc
  | 'exact' '{' exact_e '}'             // interoperation with a block expression e from Disc

ANF forms
a := x                                            // variables
  | v                                             // values
  | '!' a                                         // negation
  |  x '[' a ']'                                  // projections
  | a binop a                                     // binomial operations
  | '(' repeat(a ',') a ')'                       // products
  | '[' a ']' | '[' repeat(a ',') a ']'           // vectors
  | 'head' '(' a ')' | 'tail' '(' a ')'           // vector operations
  | 'push' '(' a ',' a ')'                        // vector operations
  | 'bern' '(' a ')'                              // Bernoulli distributions
  | 'poisson' '(' a ')'                           // Poisson distributions
  | 'uniform' '(' a ',' a ')'                     // Uniform distributions
  | 'normal' '(' a ',' a ')'                      // Normal distributions
  | 'beta' '(' a ',' a ')'                        // Beta distributions
  | 'discrete' '(' repeat(a ',') a ')'            // Discrete distributions
```

```
Binomial operations
binop := '+' | '*' | '/' | '^' | '<' | '<=' | '==' | '>=' | '>' | '&&' | '||'

Values
v := true | false                                  // booleans
  | /-?\d+\.(?:\d*|)/                               // floating-point values
  | /\d+/                                           // integers
  | '[]' | '[' repeat(v ',') v ']'                  // vectors
  | '()' | '(' repeat(v ',') v ')'                  // products
  | 'bern' '(' v ')'                                // Bernoulli distributions
  | 'poisson' '(' v ')'                             // Poisson distributions
  | 'uniform' '(' v ',' v ')'                       // Uniform distributions
  | 'normal' '(' v ',' v ')'                        // Normal distributions
  | 'beta' '(' v ',' v ')'                          // Beta distributions
  | 'discrete' '(' repeat(v ',') v ')'              // Discrete distributions
```

3.3 Interoperation

MultiPPL provides a framework in which Cont and Disc can seamlessly interoperate through boundary operators, mutually defined in each language.

An example of this is when we have components of a program which we would like to model exactly, but we would like to use this in a larger program which needs more flexibility and can be resoned about approximately. For instance, in the following program, we model a packet traversing a ladder-like network topology of of unbounded length. At each "rung" on the ladder, a unbiased node is selected for the packet to continue its traversal, and we can model each node's failure rate exactly.

```
exact fn rung (s1) {
  let route = flip 0.5 in

  let s2 = if route then s1 else false in
  let drop2 = flip 0.005 in
  let go2 = s2 && !drop2 in

  let s3 = if route then false else s1 in
  let drop3 = flip 0.001 in
  let go3 = s3 && !drop3 in

  go2 || go3
}
```

The above function models a partial traversal through this ladder network, and returns a Boolean representing whether or not the packet was able to navigate through this sub-

network without getting dropped. We want to query on the probability that a packet will successfully traverse all of these intermediate steps without getting dropped, but the network has unbounded length, which cannot be modelled exactly.

```
sample {
  ix ~ poisson(20.0);
  ix <- ix + 1;
  traversed <- true;
  while ix > 0 {
    traversed <- exact(rung(traversed));
    ix <- ix - 1;
    ()
  };
  traversed
}
```

Using Cont, we can model the length of this network using a Poisson distribution (with an average topology size of 20 rungs). We then can iterate over each subnetwork and return a sample encapsulating the success of the packet's traversal.

Evaluating this for 1000 samples, we can find the expectation of this model to be:

```
$ docker run multippl:latest --rng 1 \
         --file examples/ladder.yo --steps 1000
0.942
604ms
```

Similarly, we can use the `sample` keyword inside of a Disc program to use a Cont value in a Disc context. The MultiPPL compiler will also provide some syntactic sugar when performing variable look-ups and will attempt to perform interoperation wherever possible (as in the case of Cont's `traversed` variable binding in the above program).

For more examples of interoperation, we refer users to our submission and provided benchmarks.

4 MultiPPL Artifact Evaluation: Validation

The `multippl` compiler is responsible for providing L1 and wall-clock evaluations for an approximate inference evaluations in Fig 11 and a discrete probabilistic program evaluation in Fig 14.

4.1 Hardware Requirements

There are no explicit hardware requirements for to produce Fig 11 and Fig 14. These are able to run on commercial hardware on a single thread, but a full evaluation will take >200 hours.

Of the >200 hours 6hrs are spent evaluating the tabulated results and 200 hours are spent waiting for 400 PSI programs (100 runs in 4 evaluations) to reach a timeout of 30 minutes. Parallelizing this evaluation is not advised without large amounts of RAM, as the most expensive PSI benchmark, the ~bayesnets/alarm evaluation, takes up 17.2G per thread of residential memory. Close behind alarm is the PSI bayesnets/insurance and grids/81 evaluations, which uses ∼15G per thread of residential memory. Using less RAM than this should be acceptable on a single threaded evaluation, so long as a there is enough swap to compensate for the difference of the expected RAM.

The multippl-benchmark tool can use more threads to speed up evaluation and to reduce the size of the timeout, with PSI-specific flags to ensure PSI is still run single-threaded. Parallelizing any program using exact inference may cause programs to crash due to OOM errors. On a Thinkpad T14s Gen 3 with an AMD Ryzen 7 PRO 6850U (4.768GHz) CPU and 30G of RAM, the non-PSI portions of this benchmark can be safely run with 8 threads.

4.2 Running multippl-benchmark via Docker

The multippl-benchmark script is a multi-threaded benchmark evaluator, used to produce our evaluations. To run the multippl-benchmark command, invoke

```
$ docker run --entrypoint multippl-benchmark multippl:latest
multippl-benchmark (all|tabulate) [OPTIONS]

subcommand: all -- run all benchmarks (psi benchmarks last), then tabulate

    --num-threads NUM_THREADS Number of threads to use for non-psi benchmarks.
                              Default: 1.
    --num-runs NUM_RUNS       Number of runs to use for non-psi benchmarks.
                              Default: 100.
    --num-steps NUM_STEPS     Number of steps per run to use for non-psi,
                              approximate benchmarks. Default: 1000.

    --psi-threads PSI_THREADS Number of threads to use for psi benchmarks.
                              Default: 1.
    --psi-runs PSI_RUNS       Number of runs to use for psi benchmarks.
                              Default: 100.

    --timeout-min TIMEOUT_MIN Number of minutes before a timeout.
                              Default 30.
    --logdir LOGDIR           Directory to store execution logs.
                              Defaults to $PWD/logs.

subcommand: tabulate -- skip benchmarks and tabulate
    --logdir LOGDIR           Directory to store execution logs.
```

```
                                    Defaults to $PWD/logs.
```

The default strategy is to run 100 evaluations, single-threaded, for 1000 samples.

To save the cached files locally, outside of docker, bind to a volume to the `/data/logs` directory:

```
$ docker run -v $PWD/logs:/data/logs --entrypoint multippl-benchmark
             multippl:latest all
```

As stated above, PSI takes a considerable amount of time to produce the requisite timeouts. To reduce the running time, you may reduce the timeout duration and limit the number of runs PSI takes, relying on the provided standard error to reflect the numbers provided:

```
$ docker run -v $PWD/logs:/data/logs --entrypoint multippl-benchmark
           multippl:latest all --logdir /data/logs --psi-runs 50 --timeout-min 10
```

To speed up the non-PSI sections of the evaluation, you may increase the number of threads without parallelizing PSI processes:

```
$ docker run -v $PWD/logs:/data/logs --entrypoint multippl-benchmark
             multippl:latest all --logdir /data/logs --num-threads 8
```

If the final table is not produced, the log directory should be cleared and the benchmark should be re-evaluated. Alternatively, a partial view of the table can be generated with the `tabulate` subcommand:

```
$ docker run -v $PWD/logs:/data/logs --entrypoint multippl-benchmark
             multippl:latest tabulate --logdir /data/logs
```

4.3 Running Benchmarks Individually

To run an individual benchmark, you must first drop into an interactive zsh or bash shell:

```
$ docker run -it --entrypoint zsh multippl:latest
```

From here, you can `cd` into the `./multippl-source/bench` folder which contains the `bench.py` and `avg.py` scripts for program execution and tabulation of a single experiment.

Additionally, `runall.sh` is the source file for `multippl-benchmark` and `tabulate.py` is invoked to produce the final tables in the `multippl-benchmark tabulate` subcommand.

The `bench/` folder structure is as follows:

- `arrival/` contains subdirectories `tree-15`, `tree-31`, and `tree-63`.
- `bayesnets/` contains subdirectories `alarm`, and `insurance`.
- `grids/` contains subdirectories `3x3`, `6x6`, and `9x9` corresponding to the 9, 36, and 81 evaluations in Fig 11.

- `gossip/` contains subdirectories `g4`, `g10`, and `g20`

Each directory has a mainfile corresponding to the benchmarked tool:

- `main.psi` refers to the PSI program evaluated
- `main.py` refers to the Pyro program evaluated
- `main.yo` refers to a MultiPPL program with interoperation that is evaluated. We call this file `diag.yo` for the `grids` evaluations, as this specifies the collapsing strategy for interoperation.
- `cont.yo` refers to a MultiPPL program which only defines a Cont program.
- `exact.yo` refers to a MultiPPL program which only defines a Disc program.
- `truth.py` (or sometimes a secondary functionality of `main.py`) contains the derived groundtruth, used to calculate L1 distance.

Each experiment's subdirectory contains a symlink to `bench.py` in `./multippl-source/bench/`. A benchmark is run by invoking `python bench.py` in the subdirectory that generates logs in the current directory. Note that these benchmarks default to using half of the threads visible to docker and do **not** run PSI by default. For example:

```
$ docker run -it --entrypoint zsh multippl:latest
# in the docker shell
$ cd ./multippl-source/bench/arrival/tree-15
$ python bench.py --help
usage: bench.py [-h] [--psi] [--num-runs NUM_RUNS] [--num-steps NUM_STEPS]
                [--initial-seed INITIAL_SEED] [--noti] [--threads THREADS]
                [--logdir LOGDIR]

options:
  -h, --help            show this help message and exit
  --psi
  --timeout-min TIMEOUT_MIN
  --num-runs NUM_RUNS
  --num-steps NUM_STEPS
  --initial-seed INITIAL_SEED
  --noti
  --threads THREADS
  --logdir LOGDIR
```

Running `bench.py` will produce cached tables and data files in the parent directory of the runlogs.

5 Development

5.1 Nix Development and Running Benchmarks via Nix

MultiPPL uses nix's flakes for development. Using the source contained in `multippl-source.tar.gz` and a flake-enabled `nix` binary, the following commands enable nix development:

```
tar -xvzf multippl-source.tar.gz
cd multippl-source
git init
```

- `nix develop` enters a development shell.
- `nix flake check` runs `cargo nextest run` and checks our nix derivations.
- `nix build .#multippl .#multippl-benchmark .#multippl-docker` produces the `multippl`, and `multippl-benchmark` executables alongside the included docker images.
- `nix run .#multippl-benchmark -- <ARGS>` runs the `multippl-benchmark` executable

References

[1] Steven Holtzen. *Rsdd*. Northeastern Probabilistic Programming Laboratory. URL: `https://github.com/neuppl/rsdd` (visited on 12/30/2024).

[2] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. "Scaling Exact Inference for Discrete Probabilistic Programs". In: *Proceedings of the ACM on Programming Languages* 4 (OOPSLA Nov. 2020), 140:1–140:31. DOI: `10/gh4jhb`.

[3] Jacob Matthews and Robert Bruce Findler. "Operational Semantics for Multi-Language Programs". In: *ACM SIGPLAN Notices* 42.1 (2007), pp. 3–10. DOI: `10.1145/1190215.1190220`.