OOPSLA 2021 Artifact

This artifact is being submitted to support the paper “Statically Bounded-Memory Delayed Sampling for Probabilistic Streams”.

This artifact contains:

- `README.md` this document,
- `muf-oopsla2021.ova` a virtual machine containing source code, pre-built binaries, and tests,
- The source code of the compiler and runtime (`/src`) and the tests (`/tests`).

Claims Supported by Artifact

The artifact supports all claims in the Evaluation section of the paper. In particular, executing the analysis on the benchmark programs produces the results in Table 1 of the paper indicating whether each program satisfies the m-consumed and unseparated paths properties. The implementation of the type system presented in the paper is available in the file `src/compiler/analysis.ml`. In addition, all the examples can be compiled and run with delayed sampling.

Getting Started

First, import `muf-oopsla2021.ova` into your virtualization software. In our testing, we used VirtualBox 6.1.22 on macOS Big Sur. This VM is packaged in the Open Virtual Appliance format and can be imported into VirtualBox through `File -> Import Appliance`. The VM contains an installation of Debian Linux and has no particular hardware or network requirements.

Once the VM boots, it should present a shell as the root user with no password necessary (the root password is `root` in case it is ever required).

The artifact is in the `oopsla2021-artifact` directory. The `mufc` compiler is already installed.

```
$ cd oopsla2021-artifact
$ mufc --help
The muF Compiler. Options are:
  --only-check        Only run the static analysis (default false)
  --simulate <node>   Simulates the node <node> and generates a file <node>.ml (default main)
  --up-bound <int>    iteration bound for the unseparated paths analysis (default 10)
  -help               Display this list of options
  --help              Display this list of options
```

The `tests` directory contains a set of `.muf` programs corresponding to the benchmarks in the paper. You may execute the compiler on each program by supplying it as an argument, for example:

```
$ cd tests
$ mufc outlier.muf
```

Which

1. Displays the results of the static analysis,
2. Compiles the muF program to OCaml (`outlier.ml`)
3. Generates a simple simulation OCaml program (`main.ml`)
4. Builds an executable that runs the program (`outlier_main.exe`)

```
$ mufc outlier.muf
-- Analyzing outlier.muf
     x m-consumed analysis failure
     o Unseparated paths analysis success
-- Generating outlier.ml
-- Generating main.ml
ocamlfind ocamlc -linkpkg -package muf outlier.ml main.ml -o outlier_main.exe
```

You can now execute the program:

```
$ ./outlier_main.exe
gaussian (0.990099, 0.990099)
gaussian (0.996689, 0.665563)
gaussian (0.998758, 0.624845)
...
```

Note that stream processors never stop. Use `^C` to stop the execution.

You can also run `make tests` from the root of the project to compile all the examples.

The option `--only-check` only runs the analysis (without compilation). Run `make bench` to run the analysis on all the benchmarks presented in the paper (see the "Relationship with the paper" section below).

Relationship with the paper

Section 2

The muF program of Figure 1 is available in `tests/robot.muf`. The stream function `controller` uses a `lqr` function that acts as a place-holder for a linear–quadratic regulator. The analysis does not depend on the implementation of `lqr`, but the implementation requires matrix operations that are not yet supported.

Section 4

Compared to the syntax given in the paper, in the examples, our implementation requires the programmer to explicitly build symbolic term (`'a expr`) values as described in Section 4.1 using special constructs with the following API:

```
val const : 'a -> 'a expr
val add : float expr * float expr -> float expr
val mult : float expr * float expr -> float expr
val app : ('a -> 'b) expr * 'a expr -> 'b expr
val pair : 'a expr * 'b expr -> ('a * 'b) expr
val array : 'a expr array -> 'a array expr
val lst : 'a expr list -> 'a list expr
val ite : bool expr -> 'a expr -> 'a expr -> 'a expr

val eval : 'a expr -> 'a

val sample : 'a distribution -> 'a expr
val observe : 'a distribution * 'a -> unit
```

Concrete values can be obtained from symbolic values with the `eval` function (e.g., for the condition of a `if` statement), and `sample` always returns a `'a expr`.

To perform symbolic computations, the delayed sampling runtime requires some distribution parameters to be of type `'a expr` :

```
val gaussian : float expr * float -> float ds_distribution
val beta : float * float -> float ds_distribution
val bernoulli : float expr -> bool ds_distribution
```

Examples:

- `x = sample (bernoulli (0.5))` should be written `x = sample (bernoulli (const (0.5))` (see coin_outlier.muf).
- `x = sample (gaussian (0., 1.))` should be written `x = sample (gaussian (const (0.), 1.)` (see gaussian_gaussian.muf). Note that the `gaussian` construct uses a symbolic value for the mean but a concrete value for the variance.

- If `x = sample ...` then `if x then ...` should be written `if eval (x) then ...` (see `coin_outlier.muf`).

The muF compiler is a prototype focusing on the static analysis presented in the paper. These discrepancies could be addressed with a simple compilation pass that we leave for future work.

Section 7

To test the benchmarks presented in Table 1, run the analysis on the following files, or simply run `make bench`.

Paper Benchmark	Filename	m-consumed	Unseparated paths
Kalman	`kalman_normal.muf`	o	o
Kalman Hold-First	`kalman_first.muf`	o	x
Gaussian Random Walk	`kalman_generative.muf`	x	o
Robot	`robot.muf`	o	o
Coin	`coin.muf`	o	o
Gaussian-Gaussian	`gaussian_gaussian.muf`	o	o
Outlier	`outlier.muf`	x	o
MTT	`mtt.muf`	x	o
SLAM	`slam_array.muf`	x	o

The remaining examples in the `tests/` directory are valid programs but are not described in the paper.

For each example, the compiler returns the outcome of two analyses: m-consumed and unseparated paths analysis. For this experiment, the `--only-check` option is set to true. The compiler will not try to generate an executable. To execute an example, you need to first compile it, e.g.,:

```
$ mufc kalman_normal.muf
$ ./kalman_normal_main.exe
```

Step by Step Instructions

Installation

The `mufc` compiler depends on the `muf` branch of Zelus, and the runtime depends on ProbZelus. To install these dependencies see (in particular the docker files in these projects):

- https://github.com/INRIA/zelus/tree/muf
- https://github.com/ibm/probzelus

Once ProbZelus is installed, in the toplevel directory (containing the files `Makefile` and `muf.opam`), type `make init` to install the artifact from scratch.

You should now have the `mufc` compiler in your path. Run the following command to test your installation:

```
$ mufc --help
```

Writing muF programs

The best way to write a new muF test program is to follow the syntax of the examples. Every muF program is a series of `fun` (regular function) and `stream` (stream function) declarations. For example, in `tests/robot.muf`, there are four `stream` declarations: `kalman`, `controller`, `robot`, and `main`; and one `fun` declaration: `lqr`.

Every stream declaration is of the form

```
val <stream-name> = stream {
    init = <init-val>;
```

```
    step (<state-pat>, <arg-pat>) =
        <step-exp>
}
```

where `stream-name` is the identifier for the stream, `init-val` is the initial value of the stream, and `step-exp` is the expression evaluated at every iteration. `state-pat` is a pattern that binds the input state of the step function, and `arg-pat` is a pattern that binds the argument being supplied to the stream. Note that `init-val` must be a syntactic value (such as a numeric literal) or the instantiation of a stream function (`init` or `infer`) and that `state-pat` should be compatible with the type of the initial value.

The step function accepts the current state and a supplied argument to compute a pair of (new state, output value). Inside the step function, the following core constructs can be used:

- `sample`, which accepts a distribution and yields a sample from that distribution
- `observe`, which accepts a (distribution, value) pair and observes that distribution to be that value.
- `unfold`, which accepts a (stream, value) pair and yields a pair (output, new stream).

If the step function uses `sample` or `observe` it is considered a probabilistic stream.

Furthermore, these core constructs may also be used inside the initial value:

- `init`, which accepts the name of a stream declaration and creates an instance of that non-probabilistic stream.
- `infer`, which accepts a (number of particles, name of stream declaration) pair and creates an instance of that probabilistic stream.

There are a number of built-in distributions and operators such as `gaussian`. Please examine the supplied programs to see examples of these built-in operators.

Finally, as mentioned in Section *"Relationship with the paper, Section 4"*, our implementation requires the programmer to explicitly build symbolic term (`'a expr`) values.

The full API is the following:

```
type 'a expr
type 'a distribution

type ('s, 'i, 'o) muf_node =
    { init : 's;
      step : ('s * 'i -> 'o * 's); }

type ('s, 'i, 'o) instance =
    { state : 's;
      node : ('s, 'i, 'o) muf_node; }

val init : ('s, 'i, 'o) muf_node -> ('s, 'i, 'o) instance

val infer : int * ('s, 'i, 'o) muf_node -> ('s, 'i, 'o distribution) instance

val unfold : ('s, 'i, 'o) instance * 'i -> 'o * ('s, 'i, 'o) instance

val const : 'a -> 'a expr
val add : float expr * float expr -> float expr
val mult : float expr * float expr -> float expr
val app : ('a -> 'b) expr * 'a expr -> 'b expr
val pair : 'a expr * 'b expr -> ('a * 'b) expr
val array : 'a expr array -> 'a array expr
val lst : 'a expr list -> 'a list expr
val ite : bool expr -> 'a expr -> 'a expr -> 'a expr
```

```
val eval : 'a expr -> 'a

val sample : 'a distribution -> 'a expr
val observe : 'a distribution * 'a -> unit
```