CS421 - Final Project - Probabilisp

Authors

Pritham Marupaka, pritham2@illinois.edu

Zi Mo (Andy) Su, zimoms2@illinois.edu

Overview

Probabilistic reasoning is used in many domains, including financial stock prediction, intrusion detection for cyber-security, and image recognition. Probabilistic programming is a paradigm in which a user specifies parameters for their desired probability models and inference is performed automatically.

Probabilisp is a simple Lisp-like domain specific probabilistic programming language. The primary goal of Probabilisp is to make probability primitives accessible to programmers, while abstracting away the details of the mathematical computation. To this end, Probabilisp is able to represent problems that would require several lines of dense math library code with simple functions, e.g. `uniform`, `sample`, and `select`.

Implementation

Major tasks and capabilities

Probabalisp is a general, lisp-like programming language. As such, it supports general lisp programming features such as:

1. Function definitions
2. Lambda functions
3. Lists
4. Let expressions to declare local variables
5. Boolean, Integer, and Floating point declaration
6. Integer and Floating point arithmetic
7. Conditional execution via `if` and `cond` statements

It extends the basic functionality of a lisp-like programming language (such as Scheme) by adding primitives for probabilistic statements. This includes support for:

1. Constructing uniform distributions
2. Joining 2 distributions by concatenating their event sets
3. Sampling from a distribution with replacement
4. Sampling from a distribution without replacement
5. Computing the probability of an event given a distribution

The major tasks required to implement Probabalisp were:

1. A parser to convert raw text into a Probabalisp abstract syntax tree

2. An interpreter to maintain an environment, and execute code given a Probabalisp AST value
3. A library to represent probability distributions and events, as well as compute probabilities

Examples

There are three examples shown in `examples` that demonstrate the Probabilisp interpreter.

Components of the project

1. A monadic parser combinator library was implemented to lex Probabilisp tokens, and parse Probabilisp expressions. The parser's main abstraction is as follows:

```haskell
-- src/ParserCombinators.hs

newtype Parser a = Parser (String -> [(a, String)])

result :: a -> Parser a
result v = Parser $ \inp -> [(v, inp)]

bind :: Parser a -> (a -> Parser b) -> Parser b
bind (Parser p) f =
    Parser $ \inp -> concat [app (f x) inp' | (x, inp') <- p inp]
    where
    app (Parser q) = q

instance Functor Parser where
    fmap f (Parser a) = Parser (\inp -> map (\(x, y) -> (f x, y)) $ a inp)

instance Applicative Parser where
    pure = result
    Parser f <*> Parser a =
    Parser
        ( \inp ->
            [ (fn x, inp'') | (x, inp') <- a inp, (fn, inp'') <- f inp'
            ]
        )

instance Monad Parser where
    (>>=) = bind
    return = result

class Monad m => MonadOPlus m where
    zero :: m a
```

```
    (++) :: m a -> m a -> m a

instance MonadOPlus Parser where
    zero = Parser (\_ -> [])
    (Parser p) ++ (Parser q) = Parser (\inp -> (p inp) Prelude.++ (q inp))
```

2. A probabilistic primitives library was implemented based on the Probabilistic Functional Programming in Haskell paper.

```
-- src/Prob.hs

newtype Dist a = D {unD :: [(a, Probability)]}
deriving (Eq, Show)

type Probability = Float

type Spread a = [a] -> Dist a

type Event a = a -> Bool

uniform :: Spread a
uniform xx =
let count = fromIntegral (length xx) :: Float
    p = 1.0 / count
in D (map (\x -> (x, p)) xx)

(??) :: Event a -> Dist a -> Probability
(??) pred (D d) = foldr aux 0 d
where
    aux (x, p) acc
    | pred x = acc + p
    | otherwise = acc

join :: (a -> b -> c) -> Dist a -> Dist b -> Dist c
join f (D dx) (D dy) = D [(f x y, px * py) | (x, px) <- dx, (y, py) <- dy]

instance Functor Dist where
fmap f (D d) = D [(f x, p) | (x, p) <- d]

instance Applicative Dist where
pure x = D [(x, 1)]
(D f) <*> (D d) = D [(g x, p * q) | (x, p) <- d, (g, q) <- f]

instance Monad Dist where
return x = D [(x, 1)]
(D d) >>= f = D [(y, p * q) | (x, p) <- d, (y, q) <- unD (f x)]
```

```haskell
selectOne :: Eq a => [a] -> Dist (a, [a])
selectOne c = uniform [(v, delete v c) | v <- c]

selectMany :: Eq a => Int -> [a] -> Dist ([a], [a])
selectMany 0 c = return ([], c)
selectMany n c = do
(x, c1) <- selectOne c
(xs, c2) <- selectMany (n -1) c1
return (x : xs, c2)

select :: Eq a => Int -> [a] -> Dist [a]
select n = mapD (reverse . fst) . selectMany n

sampleOne :: Eq a => [a] -> Dist (a, [a])
sampleOne c = uniform [(v, c) | v <- c]

sampleMany :: Eq a => Int -> [a] -> Dist ([a], [a])
sampleMany 0 c = return ([], c)
sampleMany n c = do
(x, c1) <- sampleOne c
(xs, c2) <- sampleMany (n -1) c1
return (x : xs, c2)

sample :: Eq a => Int -> [a] -> Dist [a]
sample n = mapD (reverse . fst) . sampleMany n
```

3. The Probabilisp interpreter is a subset of the Scheme interpreter. The interpreter supports:

    - Functions
    - Lambda expressions
    - Integer and Float arithmetic
    - List processing functions such as `length` and `sort`
    - Conditional evaluation via `cond`

Project Status

Overall the main components of the project were implemented and are working. This includes the parser combinators and probabilistic primitives libraries. There are features and enhancements that could be added, which are highlighted below.

What works well

The supported functionality of the **Probabilisp interpreter** is highlighted below. Each feature has associated tests.

- The parser supports:
    - Arithmetic operators

  - Boolean operators
  - Comparison operators
  - List operators
  - Unary operators
  - Type predicates
  - Function definitions and applications
  - Lambda definitions and applications
  - Conditionals
  - Let
  - Quotes
  - Probability primitives
  - Sort
- The probability primitives supported are:
  - Creating uniform distributions
  - Joining distributions by concatenating events
  - Calculating the probability of an event given a distribution
  - Selecting items from a list, with and without putting them back, to create a distribution

What works partially

- The interpreter can take only single lines; having multi-line support would be a next step

Unimplemented functionality

- Common non-uniform distributions, e.g. (Gaussian, Poisson, Student's t-distribution)
- Support for constructing arbitrary distributions
- Computing conditional probabilities
- Joining distributions using arbitrary functions (only concatenation of events is supported)
- Randomization used to approximate large distributions (see Section 4 of Probabilistic Functional Programming in Haskell)

Comparison with project proposal

Comparing the end result of the project, with our proposal, we have implemented most of the proposed functionality.

Parser Combinators

Our proposal outlined the creation of a parser combinator library, which we would use to lex and parse raw text into the Probabalisp AST. This functionality is completed in our final project, and works as expected with no limitations.

Probabalisp Interpreter

We proposed an implementation of a lisp-like language interpreter with support for various features (listed above). All of the features proposed are supported in our implementation.

Probability Primitives

We proposed the support for uniform distributions, the ability to join distributions, the ability to compute probabilities of events, and the ability to sample elements from a distribution. We support all of these features. We additionally proposed support for non-uniform distributions; we did not have the chance to implement this.

Tests

We implemented 195 test cases to test the parser, probability primitives library, and interpreter. The parser test cases were ported over from MP5, giving us ample coverage of the parsing logic. Unit tests were added for each of the probability primitives, e.g., `uniform`, `(??)`, `select`, etc. Finally, to test the entire interpreter we added three test cases corresponding to the three examples in the `examples` directory. These examples cover a broad range of functionality, and having them as part of our tests gives us confidence that the interpreter is working as expected.

The tests use the Haskell Test Framework with HUnit. They are easily readable, which helps document the supported functionality of the Probabilisp language.

See Appendix A for a sample test run.

Listing

- `project/app/Main.hs`
  - The Probabilisp REPL.
  - Ability to add Probabilisp commands that will run prior to the REPL (useful for testing).
- `project/src/Core.hs`
  - Core data structures taken from MP5. We added a `Val`-type constructor, `Dist [(Val, Float)]`, which represents a distribution.
- `project/src/Eval.hs`
  - Eval functions for evaluating parsed input taken from MP5. We added:
    - `uniform`
      - Construct a uniform distribution.
      - Example:

```
(uniform '(1 2 3))
#<dist[(1,0.33333334),(2,0.33333334),(3,0.33333334)]>
```

* concatP
  - Join two distributions by concatenating their events in a list.
  - Example:

```
(concatP (uniform '(1 2 3)) (uniform '(5 6)))
#<dist[((1 . 5),0.16666667),
       ((1 . 6),0.16666667),
       ((2 . 5),0.16666667),
       ((2 . 6),0.16666667),
       ((3 . 5),0.16666667),
       ((3 . 6),0.16666667)]>
```

* ??
  - Probability of an event (predicate) given a distribution.
  - Example:

```
(?? (lambda (x) (> x 1)) (uniform '(1 2 3)))
0.6666667
```

* select
  - Select from a list without putting back.
  - Example:

```
(select 2 '(1 2 3))
#<dist[((2 1),0.16666667),
       ((3 1),0.16666667),
       ((1 2),0.16666667),
       ((3 2),0.16666667),
       ((1 3),0.16666667),
       ((2 3),0.16666667)]>
```

* sample
  - Select from a list and put back.

```
(sample 2 '(1 2 3))
#<dist[((1 1),0.11111112),
       ((2 1),0.11111112),
       ((3 1),0.11111112),
       ((1 2),0.11111112),
       ((2 2),0.11111112),
       ((3 2),0.11111112),
       ((1 3),0.11111112),
       ((2 3),0.11111112),
       ((3 3),0.11111112)]>
```

* sort
  - Sort a list.

```
(sort '(2 7 1 3 9 2))
(1 2 2 3 7 9)
```

- `project/src/ParserCombinators.hs`
  - Parser combinators based on the Monadic Parser Combinators paper.
- `project/src/Prob.hs`

  - Probability primitives based on the Probabilistic Functional Programming in Haskell paper.

- `project/src/Runtime.hs`
  - Runtime taken from MP5.
- `test/Spec.hs`
  - Tests (see Tests for more details).
- `examples/`
  - Three examples in the Probabilisp language. These can be copied into the Probabilisp REPL (`make run`).

Appendix A

Sample test run.

```
> project git:(main) $ make test
stack test
probabilisp-0.1.0.0: unregistering (dependencies changed)
probabilisp> configure (lib + exe + test)
Configuring probabilisp-0.1.0.0...
Warning: 'extra-source-files: ../README.md' is a relative path outside of the
source tree. This will not work when generating a tarball with 'sdist'.
probabilisp> build (lib + exe + test)
Preprocessing library for probabilisp-0.1.0.0..
Building library for probabilisp-0.1.0.0..
[1 of 6] Compiling Prob
[2 of 6] Compiling Core
[3 of 6] Compiling ParserCombinators
[4 of 6] Compiling Eval
[5 of 6] Compiling Runtime
[6 of 6] Compiling Paths_probabilisp
Preprocessing test suite 'spec' for probabilisp-0.1.0.0..
Building test suite 'spec' for probabilisp-0.1.0.0..
[1 of 2] Compiling Main
[2 of 2] Compiling Paths_probabilisp
Linking .stack-work/dist/x86_64-osx/Cabal-3.2.1.0/build/spec/spec ...
Preprocessing executable 'probabilisp' for probabilisp-0.1.0.0..
Building executable 'probabilisp' for probabilisp-0.1.0.0..
[1 of 2] Compiling Main [Runtime changed]
[2 of 2] Compiling Paths_probabilisp
Linking .stack-work/dist/x86_64-osx/Cabal-3.2.1.0/build/probabilisp/probabilisp ...
probabilisp> copy/register
Registering library for probabilisp-0.1.0.0..
probabilisp> test (suite: spec)
```

```
Runtime:
  arith:
    +: [OK]
    -: [OK]
    *: [OK]
    + arith: [OK]
    - arith: [OK]
    * arith: [OK]
  bool:
    and 1: [OK]
    and 2: [OK]
    and 3: [OK]
    and 4: [OK]
    and 5: [OK]
    and 6: [OK]
    and 7: [OK]
    or 1: [OK]
    or 2: [OK]
    or 3: [OK]
    and: [OK]
    or: [OK]
  comp:
    <: [OK]
    >: [OK]
    <=: [OK]
    >=: [OK]
    =: [OK]
    < comp 1: [OK]
    < comp 2: [OK]
    > comp 1: [OK]
    > comp 2: [OK]
    = comp 1: [OK]
    = comp 2: [OK]
  list:
    car: [OK]
    cdr 1: [OK]
    cdr 2: [OK]
    cons 1: [OK]
    cons 2: [OK]
    car cons 1: [OK]
    car cons 2: [OK]
    cdr cons: [OK]
    list 1: [OK]
    list 2: [OK]
    cons list: [OK]
```

```
unary:
  not 1: [OK]
  not 2: [OK]
  not 3: [OK]
  not 4: [OK]
  not 5: [OK]
  not 6: [OK]
  not 7: [OK]
  not 8: [OK]
  not 9: [OK]
  not 10: [OK]
equality:
  =: [OK]
  eq?: [OK]
  = 1: [OK]
  = 2: [OK]
  = 3: [OK]
  = 4: [OK]
  = 5: [OK]
  = 6: [OK]
  = define 1: [OK]
  = define 2: [OK]
  = lambda: [OK]
  eq? 1: [OK]
  eq? 2: [OK]
  eq? 3: [OK]
  eq? 4: [OK]
  eq? define: [OK]
  eq? lambda 1: [OK]
  eq? lambda 2: [OK]
modulo:
  modulo 1: [OK]
  modulo 2: [OK]
  modulo 3: [OK]
  modulo 4: [OK]
  modulo 5: [OK]
  modulo 6: [OK]
typepred:
  symbol? 1: [OK]
  symbol? 2: [OK]
  symbol? 3: [OK]
  symbol? 4: [OK]
  symbol? 5: [OK]
  list? 1: [OK]
  list? 2: [OK]
  list? 3: [OK]
```

```
  list? 4: [OK]
  list? 5: [OK]
  pair? 1: [OK]
  pair? 2: [OK]
  pair? 3: [OK]
  pair? 4: [OK]
  pair? 5: [OK]
  number? 1: [OK]
  number? 2: [OK]
  number? 3: [OK]
  number? 4: [OK]
  number? 5: [OK]
  number? 6: [OK]
  boolean? 1: [OK]
  boolean? 2: [OK]
  boolean? 3: [OK]
  boolean? 4: [OK]
  boolean? 5: [OK]
  null? 1: [OK]
  null? 2: [OK]
  null? 3: [OK]
  null? 4: [OK]
  null? 5: [OK]
  null? 6: [OK]
atoms:
  int: [OK]
  bool: [OK]
define:
  define 1: [OK]
  define 2: [OK]
  define 3: [OK]
  define 4: [OK]
  define 5: [OK]
lambda:
  lambda 1: [OK]
  lambda 2: [OK]
  lambda 3: [OK]
  lambda 4: [OK]
  lambda 5: [OK]
hofs:
  hof 1: [OK]
  hof 2: [OK]
  hof 3: [OK]
cond:
  cond 1: [OK]
  cond 2: [OK]
```

```
  cond 3: [OK]
  cond 4: [OK]
  cond 5: [OK]
  cond 6: [OK]
  cond 7: [OK]
  cond 8: [OK]
  cond 9: [OK]
  cond 10: [OK]
  cond 11: [OK]
  cond 12: [OK]
let:
  let 1: [OK]
  let 2: [OK]
  let 3: [OK]
  let 4: [OK]
  let 5: [OK]
quote:
  quote 1: [OK]
  quote 2: [OK]
  quote 3: [OK]
  quote 4: [OK]
  quote 5: [OK]
  quote 6: [OK]
  quote 7: [OK]
  quote 8: [OK]
  quote 9: [OK]
  quote 11: [OK]
  quote 12: [OK]
  quote 13: [OK]
  quote 14: [OK]
  quote 15: [OK]
  quote 16: [OK]
  quote 17: [OK]
  quote 18: [OK]
eval:
  eval 1: [OK]
  eval 2: [OK]
  eval 3: [OK]
  eval 4: [OK]
runtime:
  extra 1: [OK]
  extra 2: [OK]
  extra 3: [OK]
  extra 4: [OK]
  extra 5: [OK]
  extra 6: [OK]
```

```
    extra 7: [OK]
  prob:
    dice example: [OK]
    marbles example: [OK]
    cards example: [OK]
Prob:
  compress 1: [OK]
  compress 2: [OK]
  uniform 1: [OK]
  uniform 2: [OK]
  enum 1: [OK]
  (??) 1: [OK]
  (??) 2: [OK]
  join 1: [OK]
  join 2: [OK]
  prod: [OK]
  Functor:
    fmap: [OK]
  Applicative:
    pure: [OK]
    <*>: [OK]
  Monad:
    left identity: [OK]
    right identity: [OK]
    associativity: [OK]
  (>@>): [OK]
  sequ: [OK]
  selectOne: [OK]
  selectMany 1: [OK]
  selectMany 2: [OK]
  select: [OK]
  sampleOne: [OK]
  sampleMany 1: [OK]
  sampleMany 2: [OK]
  sample: [OK]

         Test Cases    Total
 Passed  195           195
 Failed  0             0
 Total   195           195

probabilisp> Test suite spec passed
Completed 2 action(s).
```