PLT: 2011.09.28
David Hu (dh2458)
Jonathan Huggins (jhh2143)
Hans Hyttinen (heh2116)
Harley McGrew (hm2457)

YAPPL: Yet Another Probabilistic Programming Language

Objectives and Motivations

Probabilistic programming languages have grown increasingly popular in recent years because they allow for the concise definition of complex statistical models. They also provide tools for sampling the (usually Bayesian) models. YAPPL is inspired by the probabilistic programming language Church, an implementation of a pure subset of Scheme (a dialect of Lisp) for generating models using probabilistic functions. Church relies on the standard Lisp syntax, which is unintuitive and difficult to read. The syntax of YAPPL is inspired by OCaml and contains special constructs for the probabilistic elements of the language, which makes it more approachable and human-readable than Church.

HANSEI is a domain specific language library for ML that does implement some probabilistic functionality. YAPPL differentiates itself from HANSEI by providing clean, native syntax for the representation of stochastic functions, memoization, and conditional sampling.

Key Features

YAPPL is a strongly-typed functional probabilistic programming language. Below we outline its core features and how they contribute to the language's functionality.

Functional. Adopting the functional programming paradigm allows for the language to philosophically stay true to mathematics and to have syntax that mimics mathematical expressions. Functions are evaluated by prepending ~.

Strongly-typed. As we are staying as mathematically true as possible, weak typing is just not possible because type coercion cannot be done on mathematical expressions, functions, or variables. Strong typing also reduces ambiguities allows the programmer to have more confidence in what he/she has written before compilation.

Pure. Well, as much as possible. Functions have no side effects because the language aims to be as similar to actual mathematics as possible. The only side effects allowed are implications of I/O and random number generation. All random number generation is eventually performed via calls to the built-in function `rand`.

Probabilistic. The heart of the language involves operations and manipulations related to probability. Thus, it has built-in support for sampling, memoization, and the conditional operator.

Sampling – When the programmer defines a function that returns a numerical value, ~ is a unary operation that can be applied to the function that denotes we are taking a sample from the function.

Memoization – We can also memoize sampling; i.e., create a version of a function that returns the same value for each call with the same arguments.

Conditionals – When evaluating a function, we can condition that the return value of that function meet certain criteria. If it so happens that the criteria can never be met, we place the fault with the programmer.

Random number generation – Random number generation is built into the language, as it is such a core tenant of probability.

types (all are immutable)
`fun, int, bool, float, string, fun[], int[], bool[], float[], string[]`

assignment and function definition
`=`

memoized function definition
`:=`

condition (predicate) definition
`= <expr> | <cond-expr>`

memoized condition (predicate) definition
`:= <expr> | <cond-expr>`

return value keyword (used in <cond-expr>):
`@`

sample, evaluate a function
`~`

arithmetic operators
`+ - * / = != < > <= >= %`

boolean operators
`or and not`

list operators
`:: @`

string operators
`^`

Example Code

```
# single line comment
### multi
    line
      comment
###

# value definition
float:q = .9;
# q is defined as 0.9 in the global scope
# sample binding;
int:x = ~geom q | @ > 5;
# x is bound to the return value of the function geom evaluated with
# parameter q; the “| @ > 5” means that the return value of ~geom q is
# conditions being greater than 5

# function definition
fun int oneOrTwo float:q = geom q | @ = 1 or @ = 2;
# a function that samples from geom q, conditional on the sample being 1 or 2

# function equivalent to above
fun int oneOrTwo2 int:q =
   int:x = ~geom q in
   if x = 1 or x = 2 then
      x
   else
      ~oneOrTwo2 q;

fun bool gtFive int:x = x > 5;

# this is a memoized function
fun int f int:n := ~geom .9 | gtFive @;
~f 0;
-> 16
~f 1;
-> 6
# later, ~f 0 still returns 16
~f 0;
-> 16

fun int[] apply (fun int int):f int[]:a =
      match(a) with
            case x :: rest -> ~f x :: ~apply f rest
            case []        -> ();
# apply has type fun int[] ((fun int int) int[])
# f has type fun int (int)
```