Odds

Opposing Discrete and Definite heuriStics

Alexandra Medway, Alex Kalicki, Daniel Echickson, Lilly Wang
afm2134, avk2113, dje2125, lfw2114

Philosophy & Motivation

"I see your boundless form everywhere, the countless arms, bellies, mouths, and eyes; Lord of All, I see no end, or middle or beginning to your totality" - Arjuna to Krishna, Bhagavad-Gita.

As programmers, we are often forced to think and program in terms of definite binaries: 0 or 1, if-else, do-while, one answer or some finite number of answers. The real world, however, is not so determinate or discrete. The real world is more fluid. The real world operates on chance and spectrums of possibility, not on simple binaries. As Arjuna remarks, the problems we seek solutions to frequently have no apparent temporal beginning, middle, or end. They must instead be conceived of in their totality. We understand this to be the programmer's job.

The programmer must take real-world problems - problems that present themselves as neither obviously discrete nor definite - and come up with solutions that can be computed on machines that operate within the realm of the discrete and definite. We understand the programmer to be a translator of sorts, from the uncertainty of the real to the general certainty of the virtual. The motivation for *Odds* is to ease this process of translation. We recognize the need to be able to compute not only on definite values, but also on discrete distributions and continuous ranges of numbers. In implementing these structures as an essential part of *Odds*, we hope to create a programming language that more seamlessly reflects the manner in which problems and solutions are posed in the real world, that is, the world of fluidity and uncertainty.

Language Description

Odds is a functional programming language that uses C-like syntax. *Odds* focuses on mathematical distributions and expresses operations on them in a simple and discrete way.

Distributions support standard operations such as addition and multiplication. In addition to these simple operations, users have the option of sampling the distribution in order to apply complex calculations on portions of the data. For example, this will allow the user to create simulations on ranges of data with a "Monte Carlo" approach. To create and define these distributions, users apply a density function to the values within a specified domain or range. If the user does not apply a density function to a range, then the distribution is assumed to be uniform.

Odds supports a number of data primitives: numbers, strings, and distributions. Once a variable is declared as one of these primitives, it is immutable. Users are given the option to pass and apply functions to these various data types.

Because users may want to process multiple items at once, *Odds* also includes lists as a collection type. Additionally, this language includes conditionals and looping. However, because *Odds* takes a functional approach, loops are discouraged in practice. Lists allow the user to store any collection of primitives or functions. Functions can be applied to the list in order to filter, modify, or add to its contents.

Code blocks will consist of variable declarations, expressions, and function calls. It will be the compiler's job to determine whether all variables are being used in scope and are being applied appropriately.

Syntax Overview

Basics

```
/*
 * An imperative sequence of statements, including variable declarations,
 * assignments, conditional statements, operations, and function calls.
 * Multi-line comments are used like this.
 */

// Single-line comments are also allowed. Variables are declared as follows:
bool myBoolean = true;
int myInteger = 0;
float myFloat = 1.0;
string myString = "Hello, World!";

// Basic arithmetic operations are allowed
bool test1 = myInteger < 5;           // true
bool test2 = !test1;                  // false
int myInteger2 = myInteger * 2 + 5;   // 5
int myInteger3 = myInteger2 ** 2;     // 25

// Basic mathematical constants are built in
PI;   // 3.141592…
EUL;  // 2.71828...

// Basic lists of ints or floats are formed with C-like array syntax
list myList1 = [0, 1, 2, 5, 10];
list myList2 = [0.0, 5.5, 7.82];

// Lists can be queried for specific elements, but are immutable once created
float myElement = myList2[1];  // 5.5
myList1[0] = 5;                // ERROR!

// Operations between lists operate pair-wise on their individual elements
myList1 + myList1;  // [0, 2, 4, 10, 20]
myList1 * myList1;  // [0, 1, 4, 25, 100]

// Standard library functions allow calculation of common
// statistical properties
mean(myList1);
stdv(myList1);
stderr(myList1);
length(myList1);
```

Functions

Functions are considered normal entities in *Odds*, so they can be assigned to variables in addition to being passed as arguments or returned from other routines. The *func* keyword indicates that a function block is about to follow:

```
// func identifier(params): return_type
func add_ten(int x): int {
  return x + 10;
}

func wrapper(): func {
  return add_ten;
}

/*
 * A main() method is required as the entry-point for the program. The return
 * indicates success (0) or failure (not 0)
 */
func main(): int {
  int x = 5;
  int y = add_ten(x);           // y == 15
  func my_add_ten = wrapper();
  int z = my_add_ten(y);        // z == 25;
}

```

Distributions

The novel portion of our language comes not from the above definitions, which closely mirror C-like syntax, but from a new type meant to simplify the process of dealing with continuous ranges of numbers or sequences of integers. To approach this problem, we introduce the new *dist* data type. The syntax was designed to approximate math style range syntax, where you declare a probability distribution to be applied over a domain of numbers.

```
// unweighted continuous distribution over domain from 2-4.
dist uniform = <2, 4>;

// unweighted discrete number range from 2-4 with step of 1 (2, 3, 4)
dist discrete1 = <2, 4, 1>;

// unweighted discrete range from 0 to 1 with step of 0.25 (0, .25, .5, .75. 1)
dist discrete2 = <0.0, 1.0, .25>;

// continuous distribution over 2-4 weighted by density function f
dist weighted1 = f | <2, 4>;

// discrete distribution over numbers from 2-4 with a step of 1, weighted by
// density function g
dist weighted2 = g | <2, 4, 1>;

/* dist operators */
int start = start(uniform);    // start == 2
float end = end(discrete2);    // end == 1.0
dist shifted = uniform + 4;    // shifted == <6, 10>
dist stretched = uniform * 5;  // stretched == <10, 20>

// Fancier: weighted3 == h | <4, 16> where h is the joint probability
// distribution of f and f
dist weighted3 = weighted1 * weighted1;

/*
 * Sample operators - if a discrete distribution, returns numbers defined as
 * one of the step values. if a continuous dist, returns floats
 * between min and max
 */

// sample one number from the distribution
int sample1 = discrete1<1>;  // sample1 == 3
float sample2 = uniform<1>;  // sample2 == 2.27

// sample multiple numbers from the distribution, returning a list
// of the samples
list sample3 = discrete2<3>;  // sample3 == [0.5, 1.0, 0.75]
list sample4 = uniform<4>;    // sample4 == [2.15, 3.7, 0.62, 1.489]
```

Code Example

Example of code in *Odds*, demonstrating calculation of an approximate area of a room.

```
func main(): int {
  // measurement of the room is in inches with 2 inches of error
  dist width = <4, 8>;
  dist length  = <6, 10>;

  // calculate the area
  dist area = width * height;    // f | <24, 80>
  int minimum = min(area);       // 24
  int maximum = max(area);       // 80
  int expectedValue = ev(area);

  return 0;
}
```

Example of code in *Odds*, demonstrating a potential Monte Carlo simulation.[1]

```
/*
 * Create normal distributions for the three variables needed for the flow
 * transfer equation. Each samples 4 standard deviations from the mean on
 * either side, capturing 99.8% of the data.
 */
func main(): int {
  dist diameter = normal(0.8, 0.003) | <0.8 - 4 * 0.003, 0.8 + 4 * 0.003>;
  dist strokeLength = normal(2.5, 0.15) | <2.5 - 4 * 0.15, 2.5 + 4 * 0.15>;
  dist rpm = normal(9.549, 0.17) | <9.549 - 4 * 0.17, 9.549 + 4 * 0.17>;

  // sample distributions 100,000 times to retrieve experimental data
  list d = diameter<100000>;
  list l = strokeLength<100000>;
  list r = rpm<100000>;

  // run Monte Carlo simulation on transfer equation
  list simulation = flow(d, l, r);
  float average = mean(simulation);
  float standardDeviation = stdv(simulation);
  float range = max(simulation) - min(simulation);

  return 0;
}

/* transfer equation for data simulation */
func flow(list d, list l, list rpm): list {
  return  PI * (d / 2) ** 2 * l * rpm;
}

/* normal distribution */
func normal(float mean, float sd): func {
  return func(float x) : float {
    float exp = -1 *  (((x - mean) ** 2) / (2 * sd) ** 2);
    return 1 / (sd * (2 * PI) ** (1/2)) * EUL ** exp;
  }
}

```

[1] Example problem and values for the normal distributions proposed in external article: http://goo.gl/Z1vlG0.