Probabilistic Functional Programming

Donnacha Oisín Kidney

July 6, 2018

How do we model stochastic and probabilistic processes in programming languages?

The same way we model any other process: using the semantics and features built into the language.

```python
from random import randrange

def roll_die():
  return randrange(1,7)
```

Problem: semantics are unclear.[1]

```
int getRandomNumber()
{
  return 4; // chosen by fair dice roll.
            // guaranteed to be random.
}
```

[1] Randall Munroe. *Xkcd: Random Number*. en. Title text: RFC 1149.5 specifies 4 as the standard IEEE-vetted random number. Feb. 2007. URL: `https://xkcd.com/221/` (visited on 07/06/2018).

```python
randomly_chosen = roll_die()

def roll_die_2():
    return randomly_chosen
```

What's the difference between `roll_die` and `roll_die_2`?

Problem: not as powerful as we'd like.

```
def expect(predicate, process, iterations):
    success = sum(predicate(process())
                  for _ in range(iterations))
    return success / iterations

expect(lambda y: 5 == y, roll_die, 100)
0.17
```

Solution: design a DSL for probabilistic programs which solves the problems above.

Three questions for this DSL:

- Why should we implement it? What is it useful for?
- How should we implement it? How can it be made efficient?
- Can we glean any insights on the nature of probabilistic computations from the language? Are there any interesting symmetries?

The first approach[2] starts with a simple and elegant answer to the second question.

We'll model a distribution as a list of events, with each possible event tagged with its probability.

$$\textbf{newtype}\ \mathit{Dist}\ a = \mathit{Dist}\ \{\mathit{runDist} :: [(a, \mathit{Rational})]\}$$

The die now looks like this:

$$\mathit{die} = \mathit{Dist}\ [(1, \tfrac{1}{6}), (2, \tfrac{1}{6}), (3, \tfrac{1}{6}), (4, \tfrac{1}{6}), (5, \tfrac{1}{6}), (6, \tfrac{1}{6})]$$

[2] Martin Erwig and Steve Kollmansberger. "Functional Pearls: Probabilistic Functional Programming in Haskell". In: *Journal of Functional Programming* 16.1 (2006), pp. 21–34. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796805005721. URL: http://web.engr.oregonstate.edu/~erwig/papers/abstracts.html%5C#JFP06a (visited on 09/29/2016).

To turn this representation into a DSL, we can use a popular abstraction: monads. These will allow us to use do-notation, a syntax for writing imperative-looking programs:

$$
\begin{array}{l}
\mathit{addPair} :: \mathit{Dist\ Integer} \rightarrow \mathit{Dist\ Integer} \\
\mathit{addPair}\ \mathit{dist} = \mathbf{do} \\
\quad x \leftarrow \mathit{dist} \\
\quad y \leftarrow \mathit{dist} \\
\quad \mathit{return}\ (x + y)
\end{array}
$$

For this probabilistic language, we need to describe the semantics of assignment ($\leftarrow$ in the example above).