Вероятностное функциональное программирование на языке Haskell.

1. Введение

В основе функционального программирования лежит принцип прозрачности по ссылками, в частности, означающий что функция f примененная к значению x всегда возращает одно и то же значение $y = f(x)$. Этот принцип, казалось бы, нарушается, если рассматривается применение функций для описания вероятностных событий, таких как метание костей: совершенно не ясно, каков будет результат и не гарантируется, что то же значение будет при повторных вызовах. Однако, два этих, казалось бы несовместимых, подхода могут быть согласованы, если вероятностные величины помещены в типы данных.

В этой статье мы продемонстрируем такой подход описав библиотеку Вероятностного Функционального Программирования (ВФП) для языка Haskell. Мы покажем, что предложенный подход не только облегчает вероятностное програмирование в функциональных языках, но, в частности, может вести к очень выразительным программам и симуляторам. Главное приемущество нашей системы в том, что симуляция могут быть описаны независимо от метода их исполнения. То есть, мы можем или полностью описать, или сделать полностью случайной любую симуляцию без изменения программного кода, который ее определяет. Ниже мы представим определения большинства функций, но опустим некоторые детали всвязи с краткостью изложения. Эти детали должны быть достаточно очевидны чтобы их мог заполнить читатель. В любом случае, все описания функций могут быть найдены в коде библиотеки доступной по адресу `eecs.oregonstate.edu/~erwig/pfp`.

Вероятностных подход в функциональном программировании основан на типе данных представляющем *распределение*. Распределение представляет исходы вероятностного процесса как набор возможных значений, с отмеченными вероятностями.

...`Probability`...

...`Dist`...

Это представление приведено здесь только для примера; оно полностью скрыто от пользователя библиотеки средствами функций, которые

конструируют и оперируют распределениями. В частности, все функции для построения значений распределений вводят ограничение, что сумма всех вероятностей непустого распределения равна 1. Таким образом, любое значения типа `Dist`представляет из себя полный набор возможных значений какого-то вероятностного события или «эксперимента».

Распределения могут представлять события, такие как вращение кости или переворачивание монеты. Мы можем строить эти распределения из списка значений используя *функции распределения*, то есть функции следующего типа

```
...Spread...
```

Библиотека определяет функции распределения для разных широко известных распределений, таких как равномерное или нормальное, и функцию `enum`, позволяющую пользователю привязать конкретные вероятности к значениям. С функцией описывающей равномерное распределение мы можем, например, описать результаты метания кости

```
...die...
```

Вероятности могут быть извлечены из распределений с помощью функции `??`, параметризуемую *событием*, которое представляется предикатом на значениях распределения.

```
...Event...
```

Существует два принципиально разных пути объединения распределений: если распределения независимы , мы можем получить требуемый результат формируя все возможные комбинации значений и перемножая соответствующие вероятности. Из соображений эффективности мы можем провести нормализацию (объединение одинаковых вхождений величины). Функция нормализации будет рассмотрена далее.

Примером объединения независимых распределений является метание нескольких костей. Функция `certanly` строит распределение из одного элемента с вероятностью 1.

С другой стороны, если второе событие зависит от первого, оно должно представляться функцией, принимающей в качестве параметра значения распрелеления первого события. Иначе говоря, тогда как

первое событие может быть представлено значением `Dist a`, второе должно быть функцией типа `a` → `Dist b`. Эта комбинация зависимых событий есть не что иное как оператор связки, если мы будем рассматривать `Dist` как монаду.

Функции `return` и `fail` могут быть использованы для описания исходов которые либо точно призойдут, либо не могут произойти, соответственно. Мы также используем синонимы `certain` и `impossible` для этих двух операций. Кроме этого, нам понадобится монадная композиция двух функций или их списка.

Мы определили `Dist`как объект типа `MonadPlus`, но это определение не важно для данной публикации.

Замечание, что вероятностные распределения формируют монаду не является новым (Giry M., 1981). Однако, предыдущие работы в основном были посвящены расширениям языков, предлагая вводить вероятностные выражения как примитивы и определяя подходящую семантику (Jones, C. and Plotkin, G. D., 1989; Morgan, C. and McIver, A. and Seidel, K., 1996; Ramsey, N. and Pfeffer, A., 2002; Park, S. and Pfenning, F. and Thurm, S., 2004). Эти книги в основом уделяют внимание определению необходимой семантики, поддерживающей некоторые аспекты, такие как эффективное вычисление матожиданий в (Ramsey, N. and Pfeffer, A., 2002) через использование монады вероятностных мер, или рассмотрению непрерывных распределений в дополнение к дискретным используя функции распределения как часть базовой семантики (Park, S. and Pfenning, F. and Thrun, S., 2004) (принося в жертву возможность представлять матожидания). Однако, нам не известно о других работах, посвященных дизайну вероятностной и симуляционной библиотеки, основанном на этой концепции.

Определение распределений через монады позволяет нам определять функции для последовательного выбора элементов из набора без возврата их назад, что приводит к зависимости поздних выборов от более ранних. Сперва определим две функции, которые в дополнение к выбранному элементу возвращают набор без него.

...`select`...

Поскольку в большинстве приложений элементы остающиеся в наборе не представляют интереса, полезно вывести функции, которые просто отображают на значения распределений. Реализация показывает

что `Dist` тоже является функтором. Также мы будем использовать имя функции `mapD` для обращения к `fmap`, чтобы подчеркнуть, что отображение идет на распределениях.

С помошью `mapD` мы теперь можем определять функции для последовательного выбора элементов из набора. Заметим, что в функция `fst` используется в `select` потому, что `selectMany` возвращает пару из списка выбранных элементов и список оставшихся (невыбранных). Мы хотим отбросить последний [список]. Мы обращаем полученный список из-за того, что элементы полученные в `selectMany` последовательно присоединяются к списку результатов, что приводит к тому, что первый выбранный элемент будет последним в этом списке.

С этим начальным набором функций мы уже можем заняться многими проблемами, которые можно найти в книгах по теории вероятности и статистике и решить их, определяя и применяя вероятностные функции. Например, какова вероятность того, что выпадет по крайней мере две шестерки при четырех бросках четырех костей? Мы можем вычислить этот результат с помошью следующего выражения.

...expr...

Примером другого класса часто встречающихся вопросов является следующий «Какова вероятность вытягивания красного, зеленого и синего шаров (в этой последовательноси) из кувшина содержащего два красных. два зеленых и один синий шар, не возвращая их назад?». С помошью перечисления для шаров, содержащего конструкторы `R`, `G` и `B`, мы можем получить результат следующим образом

...expr...

Дополнительные примеры могут быть найдены в дистрибутиве библиотеки.

Последнее понятие, содержащееся во многих примерах, это понятие вероятностной функции, то есть, функции отображающей значения на распределения. Например, второй аргумент оператора связки является такой вероятностной функцией. Поскольку во многих случаях оказывается, что типы аргумента и результата одинаковы, мы также вводим понятие перехода, которое является вероятностной функции с совпадающими типами результата и аргумента.

Обычная операция для преобразований — это применение их к распределениям, полученным через оператор связки.

В следующих двух разделах мы проиллюстрируем использование базовых библиотечных вызовов двумя примерами, демонстрирующих высокоуровневый декларативный стиль вероятностного функционального программирования. В разделе 4 мы покажем, как случайные симуляции укладываются в описаный подход и, в частности, как он позволяет приближениям распределений работать в ситуации экспоненциальному росту объема. В разделе 5 мы покажем как работать со следами вероятностных вычислений. Заключение в разделе 6 завершает статью.

2. Задача Монти Холла

В задаче Монти Холла игроку предлагают на выбор три двери, за одной из которых находится приз. Игрок выбирает одну дверь и затем ведущий открывает одну из оставшихся, за которой приза нет. После этого у игрока есть два варианта: не менять решение или выбрать другую дверь. Задача обсуждалась в (Hehner, 2004; Morgan, C. and McIver, A. and Seidel, K., 1996). Когда перед ними встает такая задача, большинство людей предполагает что изменение выбора не имеет значения — поскольку ведущий открыл дверь без приза, остается шанс 50/50 из двух оставшихся.

Однако, статистический анализ показывает, что игрок удваивает свои шансы на выигрыш при смене двери. Как это может быть? Используя нашу библиотеку, мы можем определить, верен ли этот анализ, и почему.

Простой подход таков: сначала предположим, что из трех дверей только одна выигрышная. Таким образом, первый выбор игрока совпадает с выигрышной дверь с вероятностью 1/3.

Если игрок выбрал выигрышную дверь и изменил выбор, то он проигрывает. Однако, если он в начале выбрал проигрышную дверь, ведущий имеет только один вариант двери, которую может открыть: вторую проигрышную. Таким образом, если игрок изменит выбор, он выиграет. Этот процесс может быть описан преобразованием над исходами.

Мы можем проанализировать вероятности победы, сравнивая `firstChoice` и применение преобразования `switch` к `firstChoice`.

Таким образом, если выбор остаентся неизменным, имеется очевидный шанс выиграть — 1/3, тогда как изменение выбранной двери приводит к выигрышу с шансом в 2/3.

Мы можем смоделировать игру более детально, представляя каждый шаг точными правилами, его регулирующими. Начнем строить структуру симуляции с самого низа. Начнем с трех дверей:

Далее, построим структуру, представляющую состояние игры через поля, содеражащие информацию о том, за какой дверью приз, какая выбрана и какая открыта.

Конечно, значения им будут присвоены не сразу, а по очереди. В начальном состоянии приз не спрятан, ни одна дверь не выбрана и не открыта. Поскольку состояние будет вычислено только тогда, когда все поля будут установлены, мы можем инициализировать их неопределенным значением.

Теперь каждый шаг может быть смоделирован преобразованием над `State`. Сначала, ведущий выберет случайную дверь и спрячет за ней приз.

Преобразование принимает некоторое значение и строит распределени тогоже типа. В этом случае преобразование `hide` получает значение `State` и строит равномерное распределение состояний — по одному для каждой двери, за которой может быть спрятан приз. Далее игрок выбирает, тоже случайно, одну из дверей:

Как только игрок выбрал дверь, ведущий открывает одну, не выбранную игроком и не содержащую приза. Это первое преобразование, которое зависит от значения `State`, которое оно получает. используя значение `s` в определении.

Далее, игрок может поменять или оставить неизменным свой выбор двери. Обе стратегии могут быть представлены преобразованиями над `State`.

Изменение означает выбор двери которая не выбрана и не открыта.

Также мы можем определить стратегию неизменного выбора, которая заключается в том, что не происходит никаких изменений и все значения остаются прежними.

Для создания преобразований, создающих распределение, состоящее ровно из одного элемента, мы вводим функцию `certanlyT`, которая преобразует фукцию типа `a → a` в `a → Dist a`

Наконец, мы определяем упорядоченный список преобразований описывающих ход игры: выбор двери для приза, выбор двери игроком,

открытие двери и выбор дальнейшей стратегии.

Вспомним, что функция `sequ` реализует композицию списка монадных функций, в данном примере — преобразований.

Как только все преобразования выполнены, если выбраная дверь совпадает с дверью с призом, игрок побеждает.

Мы можем определить функцию `eval`, проводящую игру для выбранной стратегии и рассчитывающую исходы для всех возможных вариантов.

Аналогично, мы можем определить значения обеих стратегий, вычисляя распределения.

Заметим, что представленная модель может быть расширена на четыре и более двери. Все что надо сделать — это добавить конструктор `D` в определение дверей и изменить `C` на `D` в определении списка. Третье решение данного примера будет кратко представлено в разделе 4.

3. Пример из биологии: Рост дерева

Многие биологические задачи основаны на вероятностном моделировании. Фактически, толчком для создания библиотеки послужил совместный проект с центром генной инженерии и биотехнологии Орегонского университета, в ходе которого нами была разработана вычислительная модель, объясняющая разработку микро-РНК генов (Allen et al., 2005).

Рассмотрим простой пример роста дерева. Предположим, дерево может вырасти за год от одного до пяти футов. Также предположим что возможно, пусть и маловероятно, что дерево может упасть во время урагана или в него ударит молния, что, предположим, убьет его, но не повалит. Как это можно представить, используя вероятностные функции?

Мы можем создать тип данных, описывающий состояние дерева и, если возможно, его высоту.

Далее мы можем ввести функцию преобразования для каждого состояния, в котором может находиться дерево.

Когда дерево живо, оно растет от одного до пяти футов в год. Распределим эти значения по кривой нормального распределения, чтобы крайние значения были менее вероятными.

Если в дерево попала молния, оно сохраняет свою высоту, однако если падает — мы ее отбрасываем. Мы можем объединить эти три

преобразования в одно, которое с различными вероятностями выбирает, что именно произойдет с живым деревом.

Здесь мы используем функцию `enum` для создания специфичного распределения с заданными вероятностями. Далее применяем это распределение к списку преобразований (вырасти, получить удар молнии или упасть) и получаем распределение преобразований. Функция развертки приводит распределение преобразований к обычному преобразованию.

Это преобразование впоследствии применяется к `t`, состоянию дерева, для получения конечного распределения для данного года. Используя начальное значение, такое как живое дерево нулевой высоты, мы можем выполнять симуляции модели дерева.

Для выяснения ситуации после нескольких поколений удобно иметь комбинининирующию функцию, производящую итерации указаное количество раз или пока выполняется некоторое условие. Три таких комбинатора собраны в классе итерирования, что позволяет производить перегрузку итераторов для преобразований и случайных изменений (описано далее).

Например, для вычисления распределение возможных состояний дерева через n лет, можем определить следующую функцию.

Для больших значений n вычисление полного распределенияи не представляется возможным. В таких случаях, случайная выборка значений распределения позволяет приблизить результат с различной степенью точности. Мы обсудим случайное моделирование в следующем разделе.

Учитывая тот факт что распределение быстро распростространяется на многие различные значения, мы не приводим результат тестовых вычислений.