АЙВИКА: ИМИТАЦИОННОЕ МОДЕЛИРОВАНИЕ В ТЕРМИНАХ ВЫЧИСЛЕНИЙ

Д.Э. Сорокин (Йошкар-Ола)

Представлен подход к решению задач имитационного моделирования, где главная идея заключена в том, что моделируемые активности разной природы представляются как некие абстрактные вычисления.

Подход воплощен автором в библиотеке Айвика, реализованной на двух языках функционального программирования: Haskell и F#. Это - результат шести лет работы.

Первая версия [3], на языке Haskell, является наиболее полной реализацией, а вторая [4] - менее функциональная, но зато ориентирована на массовые платформы .NET Framework и Mono. Обе реализации распространяются в открытых кодах и являются кросс-платформенными. Библиотека протестирована на операционных системах Windows, OS X и Linux.

На веб-страницах пакетов, составляющих версию для Haskell, можно найти подробную документацию к API библиотеки на английском языке. Также есть документация [5] на этом иностранном языке в формате PDF, где излагается суть метода, и приводятся различные примеры. Есть похожая документация [6] в формате PDF и для версии на языке F#, причем последняя содержит большое количество графических иллюстраций. Помимо этого, сами дистрибутивы пакетов для версии Haskell содержат многочисленные примеры моделей и численных экспериментов.

Возникает вопрос: почему языки функционального программирования? Дело в том, что функциональное программирование имеет развитый аппарат для создания таких абстрактных вычислений, комбинирования их, анализа, суждения об их свойствах и т.п.

Это отражается в том, что эти языки, особенно Haskell, имеют очень развитую систему типов. Кроме того, оба языка поддерживают специальный синтаксический сахар для создания таких вычислений. Это не значит, что такие вычисления нельзя создавать, скажем, на Java или C#, но это тот случай, где количество переходит в качество. Удобный синтаксис имеет значение.

Обе версии библиотеки позволяют не только формулировать имитационные модели, обсчитывать их, но также производить сбор данных, по ним строить отчеты в виде графиков, таблиц, что позволяет проводить быстрый анализ модели, а в случае необходимости экспортировать результаты в другие системы, например, в статистический пакет R.

Сама идея использовать вычисления для моделирования не является новой в мире функционального программирования. В качестве примера можно привести библиотеку Yampa [8] для языка Haskell, где вводится одно достаточно общее вычисление, применимое и для интегрирования дифференциальных уравнений, и для создания анимаций, и для написания игр.

Тем не менее, новизна работы автора в том, что Айвика целиком ориентирована именно на имитационное моделирование и имеет очень широкий охват известных приемов и парадигм. Предлагается систематизированный подход, где для основных задач существуют специализированные вычисления, что позволяет не только добиться эффективной имитации с точки зрения скорости выполнения, но и через типы вычислений помогает лучше понять саму природу моделируемых процессов.

Далее текст будет ориентирован в основном на версию для Haskell, как наиболее функциональную, по возможности содержа комментарии к встречающимся конструкциям языка.

В Айвике имитационная модель - это значение типа Simulation Results, что трактуется как вычисление Simulation, возвращающее в рамках заданного запуска имитации значение типа Results, содержащее результаты.

Иногда бывает так, что обобщая задачу, мы приближаемся к её решению. Здесь происходит ровно то же самое. Мы оперируем вычислением Simulation, возвращающим произвольные значения некоторого типа. Чтобы подчеркнуть это, еще пишут Simulation a, где a - некоторый переменный тип.

```
newtype Simulation a = Simulation (Run -> IO a)
```

Если читатель знаком с языком программирования Си++, то это похоже на шаблон template. В Java или C# некоторым аналогом является обобщенный тип generics.

Суть в том, что вводя две специальные функции, одна из которых создает новое вычисление из примитива, а вторая соединяет вычисление с его продолжением, мы можем строить вычисления произвольной сложности, а язык Haskell помогает нам в этом через специальную *нотацию do*, упрощая синтаксические конструкции, насколько возможно.

В Айвике предлагается несколько типов взаимосвязанных вычислений, где одни могут быть преобразованы к другим или запущены в рамках последних.

Так, для системной динамики существует вычисление Dynamics, которое интерпретируется как функция от модельного времени, определенная на всей оси времени сразу.

```
newtype Dynamics a = Dynamics (Point -> IO a)
```

Тогда определенный интеграл можно аппроксимировать значением типа Dynamics Double, т.е. функцией, возвращающей вещественные значения в точках времени.

А сам интеграл создается в рамках вычисления Simulation по заданной производной и начальному значению, что записывается на языке Haskell следующим образом:

```
integ :: Dynamics Double -> Dynamics Double -> Simulation (Dynamics Double)
```

Аппроксимация интеграла происходит во время запуска вычисления Simulation, т.е. во время запуска имитации модели.

Используя возможность языка Haskell трактовать значения Dynamics Double как числа, на основе *рекурсивной нотации do* мы можем кратко записывать системы обыкновенных дифференциальных уравнений, примерно так, как это делается в специализированных средах для системной динамики типа Vensim или ithink.

Однако, дифференциальные уравнения не являются сильной стороной Айвики. Фактически, они интегрируются даже медленно. Важен сам подход на основе абстрактных вычислений, который удалось успешно применить и к другим парадигмам имитационного моделирования, где скоростные характеристики могут быть уже иными.

Вычисление Event является близнецом Dynamics. Это тоже функция от времени, но строго синхронизированная с очередью событий. Через вычисления Event могут быть выражены как дискретные модели, управляемые событиями, так и управляемые временем. Причем, между моделями нет различий, поскольку это будет одно вычисление Event.

```
newtype Event a = Event (Point -> IO a)
```

Во время имитации вычисление Event запускается в рамках Dynamics, а

последнее - уже в рамках Simulation. Это значит, что имитация фактически становится одной функцией от запуска по заданным параметрам модели.

Говоря о параметрах, для них тоже есть отдельное вычисление Parameter, которое устроено примерно также как и Simulation, с той разницей, что параметры могут быть *мемоизированы*, иначе говоря, зафиксированы в рамках одного запуска. Для следующего запуска в рамках того же эксперимента будет происходить перерасчет параметра, что позволяет планировать эксперимент, а также проводить анализ чувствительности модели ко внешним, возможно, случайным параметрам, используя метод Монте-Карло.

```
newtype Parameter a = Parameter (Run -> IO a)
```

Генерация случайных чисел происходит на уровне вычисления Parameter. Более того, сам тип генератора мы можем трактовать как внешний параметр модели, что важно, например, если мы хотим использовать нестандартный генератор или же использовать генератор, дающий воспроизводимую последовательность чисел.

Возвращаясь к вычислению Event, оно является краеугольным в Айвике, поскольку остальные вычисления в итоге сводятся к нему, а оно сводится к Simulation через Dynamics.

Комбинируя *продолжения* с вычислениями Event, мы можем создать вычисление Process, которое подходит для моделирования дискретных процессов. Такой процесс можно приостановить, а затем запустить в другой момент модельного времени. С помощью вычисления Process можно выражать примерно такие же модели, какие можно записывать с помощью библиотеки имитационного моделирования SimPy для языка Python.

```
newtype Cont a = Cont ((a -> Event ()) -> Event ())
newtype Process a = Process (ProcessId -> Cont a)
```

Одновременно может быть запущено сотни и тысячи вычислений Process, и все они будут работать поверх очереди событий.

Характеристической является функция, которая приостанавливает текущий дискретный процесс на заданный период времени, что объявляется на языке Haskell следующим образом:

```
holdProcess :: Double -> Process ()
```

Здесь возвращается действие. Поэтому оно имеет такой тип.

На основе Process можно строить более сложные вычисления.

Так, мы можем определить поток заявок, распределенных во времени, возможно, с задержкой, как вычисление Stream.

```
newtype Stream a = Cons (Process (a, Stream a))
```

Тогда логично будет ввести функцию, которая будет преобразовывать входной поток в некоторый другой поток на выходе.

В Айвике такое вычисление называется Processor, и определенно оно так:

```
newtype Processor a b = Processor (Stream a -> Stream b)
```

Здесь мы видим, что Processor зависит уже от двух переменных типов, которые соответственно задают типы входных и выходных значений.

Тут можно провести аналогию с языками моделирования GPSS и Visual SLAM. Вычисления Stream и Processor можно интерпретировать как блоки, либо создающие, либо обрабатывающие транзакты.

Вычисления Processor можно распараллеливать и соединять последовательно,

что позволяет строить сложные и разветвленные сети очередей.

Например, следующая функция возвращает вычисление, которое моделирует параллельную работу заданных под-вычислений, например, станков:

```
processorParallel :: [Processor a b] -> Processor a b
```

Причем, поддерживается вытеснение ресурса на уровне дискретных процессов Process, что позволяет достаточно просто моделировать станки с поломками.

Что касается самих очередей, то они наряду с ресурсами и серверами определяются в виде отдельных сущностей, но с которыми работа происходит в рамках описанных вычислений. Запрос на захват ресурса или запрос на изъятие элемента из очереди – это примеры вычислений Process, которые могут быть приостановлены.

Только очередь или сервер хранят множество специальных счетчиков, которые обновляются по мере имитации, например, собирая статистику по размеру очереди. Для совсем простых случаев есть ссылки, которые тоже могут быть обновлены во время имитации.

Очереди и ресурсы могут использовать разные стратегии, такие как FCFS, LCFS, SIRO или на основе приоритетов. Более того, стратегии очереди можно задавать при распараллеливании вычислений Processor.

На основе Event и Process можно строить и другие вычисления.

Например, следующее вычисление определяет конечный автомат, синхронизированный с очередью событий:

```
newtype Circuit a b = Circuit (a -> Event (b, Circuit a b))
```

Здесь по заданному входу, возвращаем выход и следующее внутреннее состояние вычисления.

Таким вычислением можно моделировать некоторые цифровые схемы, только в качестве тактового генератора будет выступать очередь событий. Схемы могут быть рекурсивными, и для их создания можно использовать в Haskell *нотацию proc*, которая применима и для создания вычислений Processor только с той разницей, что вычисления Processor уже не могут быть рекурсивными в силу самой своей природы, но могут использовать очереди для создания циклической обработки с обратными связями.

Выше были перечислены наиболее важные типы вычислений. Есть и другие.

Хотелось бы еще особенно отметить такой момент, как связь непрерывного и дискретного моделирования. Существует в Айвике специальный тип Var для реализации переменной, которая хранит историю изменений. Эту переменную можно использовать в обыкновенных дифференциальных уравнениях как вычисление Dynamics, и также её можно обновлять из дискретно-событийной модели в рамках вычисления Event. Это наводит мост между двумя такими разными областями моделирования.

При этом интегралы можно использовать в дискретной части модели. Если же говорить более обще, то существует следующая цепочка преобразований, где вычисления из левой части могут быть представлены как вычисления из правой части без потери информации и контекста:

```
Parameter a -> Simulation a -> Dynamics a -> Event a -> Process a
```

Вычисление Event интересно еще тем, что в рамках этого вычисления можно моделировать агенты. Например, обработчики тайм-аута и таймера для состояния агента могут быть заданы как действия, определенные в рамках вычисления Event.

Тогда агенты будут синхронизированы с очередью событий.

Итого, можно сделать следующие выводы.

1. Разные парадигмы моделирования можно сочетать при таком подходе в рамках одной модели. Для каждой подзадачи будут свои типы вычислений, но они все взаимосвязаны и в итоге сводятся к вычислению Simulation.

2. Как побочный эффект от использования вычислений, мы можем повторно использовать код, поскольку здесь минимальная привязка к хранению состояния, а вычисления - это просто функции, которые можно создавать на-лету и комбинировать самым разным образом.

3. Имея минимум мест, где используется изменяемое состояние, мы имеем больше возможностей для распределенного и параллельного моделирования, что является перспективным направлением для развития библиотеки.

4. Фактически имеем встроенный в Haskell и F# предметно-ориентированный язык моделирования, что позволяет использовать в имитациях всю мощь базового языка программирования общего назначения с бесшовной интеграцией.

5. Все это вместе позволяет создавать нетривиальной сложности имитационные модели с динамически изменяющейся структурой, но ценой этого является довольно высокий порог вхождения, т.к. метод ориентирован на программирование, хотя и достаточно высокоуровневое.

Литература

1. **А. Борщев**, От системной динамики и традиционного ИМ – к практическим агентным моделям: причины, технология, инструменты, http://www.gpss.ru/paper/borshevarc.pdf
2. **И. Труб**, Объектно-ориентированное моделирование на C++: Учебный курс. - СПб.: Питер, 2006.
3. Aivika, http://hackage.haskell.org/package/aivika
4. Aivika for .NET, https://github.com/dsorokin/aivika-fsharp-ce
5. **D. Sorokin**, Aivika 3: Creating a Simulation Library based on Functional Programming, 2015, https://github.com/dsorokin/aivika/wiki/pdf/aivika-simulation.pdf
6. **D. Sorokin**, Aivika 3 User Guide: Version for .NET Framework and Mono, 2015, https://github.com/dsorokin/aivika/wiki/pdf/aivika-user-guide.pdf
7. **A. Alan B. Pritsker, Jean J. O'Reilly**, Simulation with Visual SLAM and AweSim, John Wiley & Sons, 1999
8. Yampa, https://wiki.haskell.org/Yampa