О представлении модельного времени при помощи механизмов функционального программирования (расширенная версия)*

Денис Буздалов, Александр Петренко, Алексей Хорошилов

Февраль 2019 (версия №4)

Содержание

*Расширенная версия одноимённой статьи, опубликованной в сборнике Труды Института системного программирования РАН, том 30, вып. 6, 2018, стр. 341-366. Эта версия включает более подробное описание используемых понятий функционального программирования и нацелена на более широкую аудиторию.

Введение

Контекстом работы является создание моделей архитектуры ответственных систем с целью дальнейшей проверки этих моделей на соответствие функциональным и нефункциональным требованиям (требованиям отказоустойчивости, потребления ресурсов памяти, задержек при передаче информации и др.).

Рассматриваются архитектурные модели программно-аппаратных систем управления, представляющие структуру системы и набор спецификаций поведения отдельных компонентов и каналов связи, в том числе описывающих темпоральные свойства поведения. Соответственно, рассматриваются такие характеристики моделируемой системы, как длительность выполнения тех или иных задач моделируемой системой и размер задержек по передаче информации внутри системы.

Данная работа посвящена динамическому анализу такого рода поведенческих характеристик архитектурных моделей. Проводившиеся ранее работы авторов [Buz14] [Buz16] использовали для динамического анализа *акторную модель*, рассматривая отдельные компоненты системы, наделённые поведением, как отдельные независимые общающиеся *акторы*, работающие в модельном времени. При этом были выявлены следующие недостатки системы, основывающейся на акторной модели, при использовании в динамическом моделировании поведения моделей систем:

- нелокальность акторной системы
  - акторы рассчитывают на наличие определённых акторов (например, которые умеют обрабатывать определённые сообщения), при этом не могут гарантировать или проверить это;
  - нельзя автоматизированно статически (не тестированием) оценить корректность замены одного актора, на другой в случае, если третий актор, который пользуется заменяемым, рассчитывает на какие-нибудь свойства исходного (например, рассчитывает на возможность принять и полностью обработать определённый тип сообщения);
- нетипизированность акторной системы
  - принятие или непринятие сообщения решается только во время выполнения и не проверяется при компиляции;
- слабость средств для поддержки множественных, недетерминированных, вероятностных или интерактивных параметров на входе:
  - множественные параметры требуют множественных запусков с комбинаторным количеством запусков и повторением общей работы;
  - вероятностные и интерактивные параметры требуют специальной ad hoc обработки в каждом отельном работающим с этим акторе; это приводит к смешению собственной логики актора с реализацией получения этих параметров и приводит к заселению акторов посторонней логикой;
  - недетерминизм требует или перезапусков (подобно множественным параметрам), или слишком раннего недетерминированного выбора отдельных параметров.

Акторная модель вычислений широко применяется в традиционном программировании, хотя при этом, там не учитывается специальным образом модельное время, с которым необходимо работать при моделировании поведения систем реального времени (для проверки свойств модели, связанных со временем).

Одновременно с этим, в программировании наблюдается тенденция противопоставле-

ния акторной модели вычисления функциональному программированию, которое позволяет решать (помимо других) те же проблемы, для которых была создана акторная модель. При этом, функциональное программирование обладает рядом преимуществ. Аналогично тому, как взамен акторной модели применяются техники функционального программирования в областях, не связанных с моделированием времени, возникает идея попробовать применить аналогичные техники взамен акторной модели в контексте модельного времени, необходимого для поведенческого моделирования систем реального времени.

В данной работе предлагается способ организации поведенческих спецификаций для динамического моделирования ответственных систем, который учитывает модельное время и использует технику функционального программирования для решения проблем, сопутствующих моделированию при помощи акторов.

Функциональное программирование

Введём некоторые важные понятия, которые будут использоваться и на которые мы будем описаться далее при изложении основных идей.

Для людей, хорошо знакомых с функциональным программированием в этом разделе не будет сказано ничего нового, однако могут быть уточнены термины, используемые в данной работе. Само понятие функционального программирования не является чётко определённым и интерпретируется по-разному разными специалистами, поэтому определим те основные принципы, на которых будем основываться далее и которые мы будем в этой работе называть функциональным программированием.

Общие понятия

Основным понятием в функциональном программировании является понятие *чистой тотальной функции* [Hug84] [SPJ86].

Есть несколько определений того, какие функции являются чистыми.

По одному определению, чистые функции не содержат *побочных эффектов*. То есть, единственное что такая функция делает — возвращает результат, основываясь только на своих аргументах. Это близко к понятию функции в математическом смысле.

Таким образом, чистая функция не может менять состояние переменных, считывать состояние глобальных переменных, взаимодействовать с аппаратурой (например, считывать данные с клавиатуры или выводить на экран) и подобные вещи.

Другим (эквивалентным) определением чистой функции является *ссылочная прозрачность* (referential transparency) её выражений. Выражение является ссылочно прозрачным, если в любом месте его использования его можно заменить на значение, которому равно это выражение.

Например, в данном примере кода

```
def f(x: Int): Int = x + 1

var cnt: Int = 0
def g(x: Int): Int = {
  cnt = cnt + 1
```

```
  x + 1
}
```

функции `f` и `g` возвращают одинаковый результат, но при этом `f` является чистой, а функция `g` — нет, потому что она обращается и изменяет глобальное состояние (переменную `cnt`). Два следующих выражения

```
f(4) + f(4)

// результат: 10
```

```
val x = f(4)
x + x
// результат: 10
```

эквивалентны с точки зрения конечного результата работы, при этом второе получено из первого заменой выражения `f(4)` на его значение (сохранённое в значение `x`). Так можно делать всегда для чистых функций.

Для функции `g` аналогичная замена не является эквивалентной. Возвращаемый результат обоих выражений будет одинаковым, но за счёт побочного эффекта, результирующее действие двух выражений будет разным:

```
g(4) + g(4)

// результат: 10
// cnt увеличился на 2
```

```
val x = g(4)
x + x
// результат: 10
// cnt увеличился на 1
```

Под тотальностью функций понимают, что она обязательно вернёт значение на любых наборах своих аргументов. Это подразумевает отсутствие механизма неявных исключений. Все исключительные ситуации должны быть отражены в сигнатуре функции.

Например, функция вычисления квадратного корня не всегда может получить результирующее значение. Чтобы определить её как чистую тотальную, в качестве одного из вариантов, можно это отразить в типе возвращаемого значения, например, возвращать частичный тип `Option[_]`. Таким образом, функция определена для всех своих аргументов и является тотальной.

```
def sqrt(x: Double): Option[Double]

// выполняются следующие равенства
sqrt(4.0) === Some(2.0)
sqrt(0.0) === None
```

Преобразования и оптимизации

Чистые функции позволяют в широком спектре случаем производить автоматические преобразования программ.

Мемоизация Из свойства ссылочной прозрачности следует, что можно производить *мемоизацию* функций, то есть запоминать результат вычисления чистой функции

для определённых аргументов и подставлять его для последующих вызовов этой функции с этими аргументами.

Это может позволить оптимизировать время выполнения программы, особенно для часто используемых функций. Функции с малым размером области определения можно заменить на табличные, вычисляющиеся за константное время.

Порядок вычислений Другим важнейшим следствием свойств чистых функций является то, что порядок вычислений функций (если результат одной функции не передаётся как аргумент другой) может быть произвольным.

Это даёт возможность компилятору и среде выполнения оптимизировать выполнения с точки зрения оптимальной загруженности процессора или других параметров.

Параллельность Независимые чистые функции можно не только вычислять в произвольном порядке, но также вычислять параллельно, если это позволяет аппаратура. В век бурного развития параллельных вычислений возможность автоматического распараллеливания вычислений является очень большим преимуществом.

Полиморфизм

Часто используемой особенностью функционального (но не только функционального) программирования является возможность объявления и использования *полиморфных* функций.

Особенностью функциональных языков с мощной системой типов является широкое использование *параметрически полиморфных* функций. Параметрически полиморфная функция — функция, имеющая одно описание, которая при этом работает при различных типах аргументов (а иногда, и возвращаемого значения) [Car85].

Классическим примером полиморфной функции является функция, добавляющая элемент к списку:

```
def cons[A](x: A, xs: List[A]): List[A]
```

Эта функция имеет типово́й аргумент `A` и работает для *любого* списка элементов типа `A` для *любого* `A`. Эта функция полиморфна по аргументу `A` (без ограничений).

Часто полиморфной функции важно объявлять, что она готова принимать только те типы в качестве типово́го аргумента, которые удовлетворяют определённым требованиям.

Например, функция, вычисляющая значение многочлена, полиморфна, потому что может работать с любым типов, являющимся *полем*:

```
def polynomial[A: Field](x: A): A = 2*x*x + 5*x
```

В этом примере, операторы сложения и умножения определены для любого типа `A`, являющегося полем (этому соответствует ограничение `Field` для этого типа).

Функции высших порядков

Неотъемлемой частью функционального программирования являются так называемые *функции высших порядков*. Такими функциями называют функции, которые либо принимают на вход другие функции, либо возвращают как результат другие функции (либо и то, и другое одновременно).

Свёртка Классическим примером функции высшего порядка является функция *свёртки списка*. Эта функция принимает на вход список, стартовый элемент свёртки и функцию свёртки:

```
def fold[A](xs: List[A], z: A)(f: (A, A) ⇒ A): A
```

Очень многие функции над элементами списка, возвращающие элемент того же типа выражаются через функцию свёртки. Например, функция суммирования всех элементов списка чисел выражается следующим образом:

```
def sum[A: Numeric](xs: List[A]): A = fold(xs, 0)(_ + _)
```

При помощи параметрического полиморфизма функция суммирования чисел обобщается до более общей функции свёртки списка для произвольного моноида:

```
def combineAll[A: Monoid](xs: List[A]): A =
  fold(xs, Monoid[A].empty)(Monoid[A].combine)
```

Функторы Другим классическим примером функции высшего порядка является функция `map` *применения функции к элементам структуры данных без изменения внутренней структуры*. Например, такая функция для списка будет иметь вид

```
def map[A, B](xs: List[A])(f: A ⇒ B): List[B]
```

Результатом работы данной функции будет список элементов типа `B`, где каждый элемент получен применением функции `f` к каждому элементу изначального списка `xs`. Тем самым, структура списка (в первую очередь, его длина) будет сохранена.

В программировании структуры данных, поддерживающие такую функцию называются *функторами*, хотя само понятия является более общим и пришло в программирование из теории категорий.

Примерами более сложных функторов можно привести графы, для которых можно построить другой граф, применив данную функцию к каждому элементу этого графа. В результате работы этой функции “рисунок” графа (то есть, количество вершин и связи между ними) сохранятся, изменятся только содержимое вершин.

Функторы (в том числе, их некоторые частные случаи, о которых мы будет говорить далее) являются важнейшей и фундаментальной составляющей функционального подхода к программированию.

Комбинаторы Функции высших порядков позволяют комбинировать функции с тем, чтобы получать новые функции. Одним из простейших комбинаторов такого рода является последовательный комбинатор функций:

```
def combine[A, B, C](g: B ⇒ C, f: A ⇒ B): A ⇒ C = x ⇒ g(f(x))
```

Эта функция позволяет последовательно комбинировать две функции:

```
val sqrt: Int ⇒ Double
val doubleToStr: Double ⇒ String

val c: Int ⇒ String = combine(doubleToStr, sqrt)
c(4) // результат: строка "2.0"
```

В дальнейшем мы увидим более сложные способы комбинирования функций (в том числе, последовательного).

Неизменяемые структуры данных

Так как чистые функции не могут иметь побочных эффектов, они не могут изменять структуры данных, передаваемые им в качестве аргументов.

Для описания чистых функций, схожих по назначению с функциями, изменяющими структуры данных, используются функции, описывающие новое состояние структуры данных после изменения. Описанные выше функции `cons` и `map` для списков обладают именно таким свойством: они конструируют новый список на основе старого, не затрагивая последний.

Для такого подхода “классические” структуры данных подходят плохо. Например, функции, возвращающей массив с добавленным в начало элементом, приходилось бы копировать содержимое старого массива в новый, что требовало бы линейного времени выполнения. При этом, есть структуры данных, хорошо подходящие для работы с чистыми функциями.

Так как чистые функции не могут изменять сами структуры данных, с которыми работают, можно это свойство неизменяемости распространить на сами структуры данных и получать преимущества от этого.

> Например, неизменяемые связный список (linked list) может иметь очень быстрые (константного времени) операции добавления и удаления элемента из головы, потому что два списка, имеющие общий хвост могут использовать общие участки памяти (объекты) для хранения общего хвоста (а целостность данных гарантируется неизменяемостью структуры данных).
>
> Однако, при этом, такого рода список имеет медленные (линейные) операции работы с хвостом, а также выборки элементов по индексу.
>
> Однако, существуют другие неизменяемые структуры данных, эффективные для этих операций. Например, существует структура данных, имеющая амортизированно константное время операций индексации, добавления/удаления и изменения элементов. Повторюсь, что под “добавлением”, “удалением” и “изменением” имеются ввиду функции, возвращающие новые объекты с, соответственно, добавленным, удалённым или изменённым элементами [CPrf] [Oka98].

Одним важнейшим преимуществом использования неизменяемых структур данных (помимо существенной экономии памяти при пересечениях по данным) является отсутствие необходимости в синхронизациях при использовании таких структур данных кодом из разных потоков. С учётом автоматической распараллеливаемости, упомянутой выше, это даёт существенные преимущества для автоматического ускорения работы функционального кода.

Вычисления с эффектами

Побочные эффекты

Выше говорилось, что основным способом описания вычислений в функциональном программировании являются *чистые функции*, т.е. функции без *побочных эффектов*. Однако, в конечном итоге, программы должны производить побочные эффекты — например, читать и писать в базы данных, выводить информацию пользователю, читать и писать

файлы, взаимодействовать с другими программами по сети и т.п. Таким образом, перед функциональным стилем программирования встаёт вопрос совмещения *чистоты* используемых функций и *побочных эффектов* для выполнения полезных действий.

Для того, чтобы совместить эти две, казалось бы, несовместимые вещи, в функциональном программировании используют тот факт, что *чистое* вычисление может возвращать достаточно произвольную структуру данных, в том числе структуру данных, которая внутри содержит описание вычисления, не являющегося *чистым* (т.е., *вычисление с побочным эффектом*).

При этом, если собственно запуск *вычисления с побочным эффектом* недоступен *чистому* коду, то не нарушается никаких предположений о чистом коде и всё, что говорилось о нём ранее, продолжает выполняться.

Таком образом, результатом работы чистого кода может быть описание *вычислений с побочным эффектом*, которое может быть далее выполнено внешним (по отношению к самой программе) исполнителем. Тем самым, производится разделение выполнения функциональной программы на две части:

- в первой части исходная программа, написанная на функциональном языке, вычисляет некую структуру данных, которая содержит помимо других вычислений инструкции, содержащие побочные эффекты, предназначенные для выполнения специальным исполнителем; на этом выполнение собственно программы заканчивается;
- во второй части специальный исполнитель исполняет инструкции с побочным эффектом из вычисленной специальной структуры данных.

В общем случае, может существовать несколько специальных исполнителей инструкций с побочным эффектом и, соответственно, структур данных, с которыми эти исполнители работают. Некоторые из них предназначены для выполнения произвольных внешних побочных эффектов, некоторые только для определённого ограниченного множества (что позволяет гранулировано контролировать их использование). Существуют также библиотеки, позволяющие абстрагироваться от конкретного исполнителя побочных эффектов, позволяя писать полиморфный код с указанием используемых эффектов.

Произвольные эффекты Чаще всего, структуры данных, которые инкапсулируют в себе *вычисления с побочным эффектом*, называются `IO` или `Task` [SPJ92] [CtsEff] [MxTsk] [ZIO]. Соответственно, могут быть чистые функции, которые возвращают значения этих типов, например, функции строкового ввода и вывода:

```
def printLine(s: String): IO[Unit]
def readLine: IO[String]
```

В данном случае, функция `printLine` чистая и она лишь возвращает вычисление, которое распечатает строку на экран (в возвращаемом значении типа `IO[Unit]`). Аналогично, функция `readLine` возвращает не строку, введённую пользователем, а вычисление, которое может вернуть строку, введённую пользователем.

> Важно отметить, что так как функция `readLine` чистая и при этом не имеет аргументов, она, на самом деле, константная. При этом, никаких противоречий не возникает: она возвращает одно и то же значения типа `IO`, которое, в свою очередь, инкапсулирует одно и то же *вычисление с побочным эффектом* (по считыванию строки от пользователя), и только уже это вычисление может возвращать различные результаты.

Обычно, структуры данных типа `IO` являются *функторами*, то есть могут быть преобразованы другими чистыми функциями, например,

```
readLine map { _.toIntOption getOrElse 0 }
// результат типа IO[Int]
```

Как мы увидим дальше, это свойство, а также ряд более сильных свойств, будут важны для композиции функций, возвращающих `IO`.

Гранулярные эффекты Существуют способы описания вычислений, которые *полиморфны* по конкретному типу возвращаемого значения, но имеют гранулярный контроль за производимыми эффектами.

В этом случае функция возвращает тип произвольного эффекта `F[_]`. При этом в сигнатуре функции указываются ограничения на возможные варианты эффектов, допустимые для осуществления этой функцией.

Например, существует способ доступа до текущего времени (что тоже является побочным эффектом). Такая функция возвращает тип эффекта `F[_]`, который ограничен тем, что внутри него допускается узнавать текущее время. При этом, действие, производимое возвращаемым значением, не может совершать никаких побочных эффектов, кроме указанного.

```
def isLate(time: Long): Boolean = ???

def getLate[F[_]: Clock:Functor]: F[Boolean] =
  Clock[F].realTime(MILLISECONDS) map isLate
```

В этом примере выражение `Clock[F].realTime` имеет тип `F[Long]` и описывает *вычисление с побочным эффектом*, возвращающее текущее время.

Аналогичным образом, существуют другие гранулированные и ограниченные “малые эффекты”, как, например, управление потоками выполнения или доступа к базам данных. Всё это позволяет точно и тонко определять что допускается, а что не допускается тем или иным функциям.

Композиция вычислений с эффектами

Промоделировав функции с побочным эффектом как чистые функции вида `A ⇒ IO[B]` или более гранулярно как `A ⇒ F[B]` для определённых `F`, мы получаем некие возможности обработки информации, полученной при помощи побочного эффекта, но их пока недостаточно.

Например, при помощи функций `readLine` и `printLine`, описанных выше, не получается распечатать строку, которая содержала бы информацию из считанной строки, потому что `IO[A]` — это лишь описание *вычисления с побочным эффектом*, возвращающего `A`, поэтому невозможно его получить вне `IO` в чистой функции.

Для решения проблем *композиции* функций вида `A ⇒ IO[B]` и `B ⇒ IO[C]` была применена техника, которая называется *монадами* [SPJ92] и которая имеет свои истоки в теории категорий [Mog91] [Wdl92].

Если какой-то тип данных `M[_]` является монадой, это означает, что имея `M[A]` мы можем применить к нему функцию `A ⇒ M[B]` и получим значение `M[B]`. Эта операция исторически имеет названия `bind`, `flatMap` или оператора `>>=`.

На примере с `IO[A]`, описывающей вычисление с побочным эффектом, это означает, что если мы предъявим функцию, обрабатывающую значение типа `A`, полученное в результате побочного эффекта, к функции, которая его обработает и вернёт `IO[B]`, мы сможем их скомпозировать с тем, чтобы получить `IO[B]`.

```
def f[A, B](a: IO[A], f: A ⇒ IO[B]): IO[B] = a »= f
```

Многие языки программирования имеют специальный синтаксис для длинных последовательных композиций монадических значений и функций. Например, следующие две функции `g1` и `g2` функционально эквивалентны:

```
def g1[A, B, C](ioa: IO[A], ab: A ⇒ IO[B], bc: B ⇒ IO[C]): IO[C] = for {
  a ← ioa
  b ← ab(a)
  c ← bc(b)
} yield c

def g2[A, B, C](ioa: IO[A], ab: A ⇒ IO[B], bc: B ⇒ IO[C]): IO[C] =
  a »= ab »= bc
```

При этом, в некоторых ситуациях удобен один синтаксис, в некоторых другой. Например, мы можем скомпозировать описанные выше функции `readLine` и `printLine` для простой программы-приветствия:

```
def greet: IO[Unit] = for {
  _ ← printLine("Введите ваше имя")
  name ← readLine
  _ ← printLine(s"Здравствуйте, $name")
} yield ()
```

Другой важной чертой монад является то, что всегда есть возможность по чистому значению получить монадическое, то есть получить значение как бы с *нулевым эффектом*. Обычно такого рода функции называются `pure` или `point` и имеют вид `A ⇒ F[A]`.

Монадическая композиция по сути является последовательной композицией зависимых вычислений, где каждое вычисление может производить эффект.

> Для монад типа `IO` монадическая композиция является своего рода аналогом “точки с запятой” в императивных языках программирования, где точка с запятой часто разделяет отдельные операторы, каждый из которых может обладать побочным эффектом.

Как мы увидим далее, в функциональных языках эта композиция позволяет более гибко управлять этой композицией. При этом, такого рода композиция гарантирует упорядоченность возникновения побочных эффектов.

Однако, не всегда вычисления с эффектами должны быть строго упорядоченными. Причиной могут быть как соображения *эффективности* (не всегда вычисления стоит делать последовательно), так и соображения *семантики* (не всегда нужна или определена именно последовательная композиция эффектов).

Для такого рода композиций была предложена абстракция *аппликативных функторов* [McBr08]. Аппликативные функторы позволяют слить пару эффектов в эффект получения пары объектов:

```
def product[A, B](a: F[A], b: F[B]): F[(A, B)]
```

Так как аппликативный функтор является функтором, к значению вида `F[(A, B)]` можно применить функцию вида `(A, B) ⇒ C` с тем, чтобы получить значение типа `F[C]`.

Для эффектов, которые можно выполнять параллельно, реализация аппликативного функтора может позволить выполнять эти вычисления параллельно. В общем случае, абстракция аппликативного функтора позволяет осуществлять параллельную композицию вычислений с эффектами.

Эффекты в широком смысле

На понятие *эффекта* можно посмотреть шире: под эффектом можно понимать не только побочные эффекты в виде изменения состояния внешнего по отношению к программе мира или её переменных.

В общем случае (в зависимости от потребностей точного моделирования семантики выполнения в чистом функциональном языке) под эффектом можно понимать даже обращение к значению переменной находящейся, быть может, на другом узле вычислительной сети при распределённом выполнении [Kis16]. В более частных случаях под эффектом могут пониматься любые действия функции помимо вычисления и возврата простого единственного значения, примеры которых будут рассмотрены далее.

Важным свойством функций с побочным эффектом является то, что они могут делать ещё что-то, кроме вычисления возвращаемого значения, а также пользоваться ещё чем-либо, кроме своих аргументов для своей работы. При функциональном программировании вместо *фунций с побочным эффектом* вида `A ⇒ B` рассматривают *чистые* функции вида `A ⇒ F[B]`, где `F` инкапсулирует сам эффект (например, произвольный побочный эффект, когда это тип `IO`). При более широком взгляде на понятие *эффекта* можно рассматривать функции вида `A ⇒ F[B]` как *чистые* аналоги *функций с эффектом* вида `A ⇒ B` для более широкого класса типов `F`, чем просто побочный эффект.

Рассмотрим некоторые примеры монад, которые моделируют те или иные *эффекты* в чистом функциональном программировании.

Монада произвольного побочного эффекта Упоминавшиеся ранее монады `IO` и `Task` и другие подобные позволяют описывать вычисления с произвольным побочным эффектом.

При этом, в отличие от описанных выше эффектов, не затрагивающих побочный эффект, действия, производимые такими вычислениями не отражаются явно в чистом коде. Эффекты могут содержать ввод-вывод, из-за чего возникает недетерминизм возвращаемого в эффекте результата.

Тривиальная монада Существует монада, описывающая “нулевой” эффект. Такую монаду называют монадой `Id`. Функция обладающая таким эффектом (т.е., функция вида `A ⇒ Id[B]`) на самом деле является чистой (фактически, `A ⇒ B`). Последовательная композиция функций с тривиальным эффектом совпадает с простой композицией функций.

На практике такую монаду используют в для полиморфного по эффектам кода, где в конкретном случае никакие эффекты не нужны или действие определённых эффектов моделируется чистым кодом (например, при тестировании).

Монада частичных вычислений В качестве эффекта можно рассматривать ситуацию, когда функция в некоторых случаях не может возвратить значение, то есть, функция не является *тотальной* (является *частичной*). В таких случаях используют монаду `Option` (она же `Maybe`).

В качестве примера рассмотрим чистую функцию, разбирающую строку и возвращающую число, записанное в этой строке. В императивном программировании часто используют *выбрасывание исключения* в случае, когда переданная строка не является строковой записью числа. В чистом функциональном программировании рассматривают *тотальную* функцию `String ⇒ Option[Int]`, которая возвращает либо контейнер `Some` с результатом, либо `None`, обозначающий отсутствие результата.

```
def toInt(s: String): Option[Int] = ...

toInt("123")
// результат типа Option[Int]: Some(123)

toInt("abc")
// результат типа Option[Int]: None
```

Последовательная композиция функций с эффектом частичных вычислений обладает семантикой *раннего падения* (fail fast). Это означает, что стоит любой функции в последовательности функций с этим эффектом вернуть значение типа `None` (т.е., отсутствие результата), вычисление прекращается с тем же результатом.

Монада множественных вычислений Некоторые функции вычисляют несколько значений (в том числе и отсутствие значений). Это тоже можно рассматривать как эффект и композировать такие многозначные функции друг с другом. Композицией многозначных вычислений является аналог декартового произведения (при необходимости с учётом повторений). Обыкновенный список `List` часто является подходящей монадой для многозначных вычислений.

Монада вычислений с контекстом Монада `Reader` представляет собой вычисление, которое имеет доступ до дополнительной информации: контекст или конфигурация вычисления. Возможность такого доступа (явно не отражённого напрямую в сигнатуре функции) также является эффектом (в широком смысле).

Монада вычислений с метаинформацией В качестве эффекта рассматривают формирование дополнительной информации при вычислении основного значения. Эта информация является своего рода описанием (возможно пустым), приложенным к вычисленному значению. За это отвечает монада `Writer`.

Для корректной работы этой монады, а в частности, для обеспечения композируемости монадических вычислений, нужно обеспечить, чтобы соответствующие используемые описания, которые идут вместе с вычисленным значением, также композировались.

Характерным примером использования этой монады является описание вычислений, которые параллельно с какими-то вычислениями, формируют лог событий. В этом случае, этот лог событий и будет являться описанием этих вычислений.

При последовательной композиции вычислений с логом, соответствующие логи композируются. В общем случае, операция композиции логов может быть сколько

угодно сложной (как и структура данных самого лога), но очень часто используют простые списки сообщений или даже просто строки. В случае, если в качестве лога используют список событий, композицией логов обычно является конкатенация списков.

```
def inc(x: Int): Writer[List[String], Int] = Writer(
  List(s"inc $x"), x + 1)
def mul(x: Int, y: Int): Writer[List[String], Int] = Writer(
  List(s"mul $x and $y"), x * y)

val w = for {
  a ← inc(4)
  b ← inc(5)
  c ← mul(a, b)
} yield c

w.run // результат: (List("inc 4", "inc 5", "mul 5 and 6"), 30)
```

Вероятностная монада Ещё одним примером эффекта можно рассмотреть возврат не единичного значения, а вероятностного распределения значений. При рассмотрении конечных дискретных распределений вероятности, это является обобщением эффекта множественных значений (где каждому значению приписана вероятность возникновения этого значения).

Также, можно рассматривать распределения, не являющиеся дискретными. В этом случае, результирующее вероятностное распределение должно быть далее сынтерпретировано (например, при помощи генератора случайных чисел).

Наслаивание монад

Проблема

Очень часто возникает ситуация, когда либо в одновременно используются две или более монад для одного и того же значения (то есть, код производит несколько эффектов), либо когда есть потребность композировать функции, использующие различные монады (то есть, композиция функций, имеющих разные эффекты). Уже через небольшое время после использования монад для моделирования эффектов (в широком смысле) было показано, что в общем случае разные монады не композируются [Ste94].

Также, было показано, что в общем случае порядок наслоения монад друг на друга существенно влияет на результат, что оказалось ограничением для написания полиморфного кода.

Например, композиция множественных и частичных вычислений может продуцировать (в зависимости от порядка применения эффектов множественности `List` и частичности `Option`) либо множественность частичных результатов (`List` из `Option`'ов), либо частичный множественный результат (`Option` `List`'ов).

Monad transformers

В качестве попытки разрешения этих и связанных с этим проблем, был предложен механизм *трансформаций монад* (monad transformers) [MTL95].

Однако, этот подход не позволял полноценно описывать код, который продуцирует отдельный эффект или несколько эффектов, при этом композируемый с другими эффектами. Код, использующий этот принцип, должен был всегда чётко определять в какой последовательности применяются используемые эффекты и не допускал произвольной вставки других эффектов между используемыми (если только не был написан в специальной и очень громоздкой манере).

Таким образом, раньше времени принималось решение о последовательности применяемых эффектов. Из-за этогоневозможно было описать единственную функцию, которая могла бы композироваться с другими функциями тех же эффектов в другом порядке. Также, возникали проблемы с возникновением несколько раз одного и того же эффекта в этой последовательности эффектов.

Extensible effects

Позже был предложен подход, описывающий специальную монаду, названную `Eff`, которая позволяла описывать ленивое вычисление с неупорядоченным множеством эффектов [Kis13] [Kis15]. Это неупорядоченное множество эффектов (по историческим причинам иногда называемое *стеком эффектов*) является параметром типа `Eff` и тем самым присутствует в сигнатуре всех функций, возвращающих значение типа `Eff`.

При этом, были сформулированы правила монадической композиции таких вычислений. Результирующее вычисление имеет объединение множеств эффектов композируемых вычислений. Лёгкость объединения (и дальнейшей композиции) таких вычислений обеспечивается использованием *ограниченного полиморфизма* (bounded polymorphism) по этому множеству. В таком случае, объединение множеств сводится к объединению ограничений на полиморфный параметр вычислений, что естественным образом поддерживается в языках, поддерживающих ограниченный полиморфизм.

Соответственно, для того, чтобы выполнить вычисление со всеми эффектами, требуется применить к значению типа `Eff` (имеющего множество эффектов как параметр типа) специальные интерпретаторы эффектов (в требуемом порядке), где каждый из интерпретаторов будет либо убирать из множества эффектов как минимум один, либо преобразовывать один эффект в другие. Такое должно продолжаться до тех пор, пока множество эффектов не станет пустым. В таком случае, соответственно, полученное значение будет чистым.

Этот подход позволяет гибкие возможности по реализации полиморфного по эффектам кода, который не требует раннего упорядочивания эффектов и не имеет проблем с повторением одного и того же эффекта в наслаивании несколько раз (из-за рассмотрения множества эффектов взамен последовательности). Ограничением подхода является то, что все функции, имеющие какой-либо эффект, необходимо оформлять в виде функций возвращающих объект специального типа `Eff` (с определёнными параметрами типа), а также необходимость использования примитивов, специфичных для типа данных `Eff`.

> Рассмотрим пример вычислений, которые оформлены для использования с монадой `Eff`.
>
> ```
> def multipleIntegers[R: _list]: Eff[R, Int] = ???
> def sqrt[R: _option](x: Int): Eff[R, Double] = ???
>
> def composition[R: _list:_option]: Eff[R, Double] = for {
>   i ← multipleIntegers
> ```

```
    x ← sqrt(i)
  } yield x
```

В этом примере мы имеем два вычисления (`multipleIntegers` и `sqrt`), которые полиморфны по эффекту с использованием монады `Eff` (параметр типа `R`), но при этом первая функция требует, чтобы `R` поддерживал эффект множественности, а вторая — эффект частичных вычислений. Соответственно, функция композиции требует оба этих эффекта.

Далее, мы можем сынтерпретировать функцию композиции как минимум в двух упорядочиваниях эффектов: сначала эффект множественности, затем эффект частичных вычислений или наоборот.

```
type LO = Fx.fx2[List, Option] // множество из двух эффектов

composition[LO].runList.runOption.run
// результат типа Option[List[Double]]

composition[LO].runOption.runList.run
// результат типа List[Option[Double]]
```

Также, мы можем использовать эту функцию в более обширном контексте, например, с эффектом ввода-вывода. Таким образом, мы можем использовать те же самые функции в произвольных контекстах, которые предоставляют обработку требуемых конкретному вычислению эффектов.

```
type Ext = Fx.fx3[List, Option, IO]

composition[Ext].runList.runOption
// результат типа Eff[Fx.fx1[IO], Option[List[Double]]]
```

Полиморфизм по эффекту

Своего рода обобщением предыдущих двух решений является вместо использования определённой структуры данных и полиморфизма по её параметру, использования *полиморфизма высокого порядка* (higher-kinded polymorphism) по самому типу эффекта вкупе к *ограниченным полиморфизмом* в каждой отдельной функции. По историческим причинам (и истокам в теории категорий) этот подход получил название tagless-final стиль [Kis12] [Wel17] [Nizh18], также, в одном своём частном, но широко употребимом случае, получивший название “`F` с дыркой”, что подчёркивает использование полиморфизма высокого порядка по эффекту.

В этом подходе функции, обладающие тем или иным эффектов, предназначенные для композируемости, объявляются полиморфными по типу эффекта (то есть, по типу высшего порядка, обычно обозначаемого `F`), возвращающими значение `F` с конкретным типом. При этом, для описания какие именно эффекты могут быть произведены этой функцией, используется *ограниченный полиморфизм*, то есть полиморфизм, который требует конструктивных доказательств того, что `F`, по которому осуществляется полиморфизм, обладает достаточными возможностями обеспечивать заданный эффект.

Этот подход имеет принципиально лишь то отличие от подхода с монадой `Eff`, что в общем случае для типа `F` не обязательно требуется, чтобы он являлся монадой. Если функции с эффектом `F` не требуется именно монадическая сущность эффекта, она не

обязана требовать это. С другой стороны, если же функции требуется, чтобы эффект `F` был монадой (например, сама функция является монадической композицией других функций), она должна потребовать монадичность `F` явно (в отличие от монады `Eff`, которая является монадой сама по себе). Это может иметь свои преимущества и позволяет функции чётче декларировать что именно ей требуется (ограниченный полиморфизм эксплуатируется не только для типов эффекта, но и для способа композиции).

```
def multipleIntegers[F[_]: Multiplicity]: F[Int] = ???
def sqrt[F[_]: FunctorFail](x: Int): F[Double] = ???

def composition[F[_]: Muptiplicity:FunctorFail:Monad]: F[Double] = for {
  i ← multipleIntegers
  x ← sqrt(i)
} yield x
```

Другие альтернативы

Стоит упомянуть язык Unison, функциональный язык в разработке, который использует наслоение функторов и монад на уровне языка, а не на уровне библиотек (как это делают Haskell и Scala) [Run18]. Фактически, там переопределена композиция функций, которая может быть реализована как композиция аппликативных функторов или монад (в зависимости от типа выражения). Из-за этого улучшается читаемость кода и, возможно, скорость компиляции. Однако, по всей видимости, принципиальные возможности остаются на одном и том же уровне. Поэтому, в дальнейшем мы будем рассматривать реализацию наслоения монад при помощи библиотек к широко используемым языкам (в частности, Scala).

Существует попытки контролировать эффекты при помощи введения допольнительных параметров типов к структурам данных (функторам, монадам), которые отвечают за преобразования и композицию эффектов. Это приводит к обобщениям этих структур данных (например, появлению т.н. *indexed monad*), однако требует определённой сложной машинерии на уровне языка [Orch14b].

Альтернативным вариантом решения проблемы композируемости монад является использование *вместо* монад для композиции вычислений с эффектами абстракции, являющейся чуть менее мощной, чем монада, но при этом всё ещё применимой и при этом более легко композируемой. Эта абстракция получила название *стрелки* (arrow) [Hug00], [Hug04] и по выразительной мощности является более мощной, чем аппликативный функтор, но менее мощной, чем монада [Wdl11]. Чисто функциональное программирование с эффектами не в монадическом, а в стрелочном стиле требует определённого несколько необычного (и порой непривычного) способа выражения программы, однако позволяет комбинировать различные эффекты легко (для тех эффектов, к которым применимо понятие стрелки) [Pol15], [Foy17], [JDG18].

Монадический подход к моделированию

Базовая идея

Начнём с представления модельного времени при помощи использования монад. Основной идеей “монадического” подхода к работе с вычислениями в модельном времени состоит в рассмотрении использования модельного времени как эффекта (в широком

смысле).

В простейшем случае мы можем представлять функцию, моделирующую вычисление, которое требует определённое модельное время, как функцию, возвращающую вычисленное значение вместе с величиной требуемого времени.

Для *композируемости* такого рода вычислений, представим эффект вычисления вместе с тратой модельного времени как монаду; будем называть её `TimedValue`.

Например, рассмотрим простейшую функцию сложения двух чисел.

```
def sumTimed(x: Int, y: Int): TimedValue[Int] = (x + y) |: 2.microseconds
```

Данная функция описывает вычисление, состоящее из сложения двух чисел, которое (исходя из определения этой функции) занимает две микросекунды модельного времени на вычисление. Здесь оператор `|:` — синтаксически удобный способ создания значений монады `TimedValue`.

Эта монада похожа на монаду `Writer`, но она не имеет типового параметра накапливаемого значения (тут он зафиксирован, это тип `Time`). Как следствие, эта монада не требует дополнительных ограничений на способ композиции для времени. Для последовательной композиции вычислений времена вычислений складываются.

Таким образом, имея, например, две функции

```
val inc: Int ⇒ TimedValue[Long]
val sqrt: Long ⇒ TimedValue[Double]
```

мы можем скомпозировать их в одну функцию, занимающую сумму модельного времени обеих функций:

```
inc ⟹ sqrt: Int ⇒ TimedValue[Double]
```

Последовательная и параллельная композиция

Функция сложения выбрана именно для *последовательной композиции* вычислений в модельном времени, подходящая для использования в монаде, то есть для описания зависимых вычислений.

В общем случае, может использоваться произвольная *ассоциативная бинарная* функция композиции двух времён. Выбор этой функции в большой степени определяет семантику комбинации монадических значений `TimedValue`.

Например, эта функция могла бы добавлять определённую величину времени для моделирования задержек на собственно вызов функций (при необходимости и очень точном моделировании затрачиваемого времени). Однако, в данной работе мы не будем рассматривать такой случай.

В качестве другого (весьма важного) примера альтернативной функции можно выбрать функцию максимума. При таком выборе, получится параллельная (с точки зрения модельного времени) комбинация вычислений `TimedValue`. Однако, с такой композицией может возникнуть проблема *физического смысла* результата. В частности, не всегда доступна параллельная композиция вычислений, занимающих (модельное) время, из-за нехватки (модельных) ресурсов даже в случае независимости этих вычислений по данным.

Для гранулированного подхода используются различные абстракции (например, `Parallel` [CtsPar]), которые позволяют не давать композировать функции, тратящие модельное время параллельно в случае нехватки модельных ресурсов для этого. Это достигается за счёт сложного параметрического полиморфизма, фактически эквивалентного по выразительной мощности системе Хорновских дизъюнктов [Leo14].

Монады времени в стеке монад

Как и другие монады, монаду вычислений с модельным временем можно использовать в стеке с другими монадами.

Это позволяет использовать один и тот же монадический код в различных ситуациях без переписывания и без изменения семантики с сохранением типобезопасности. При этом, различная семантика итоговой работы функций достигается за счёт различных комбинаций функций-интерпретаторов при использовании этого монадического кода.

Правда, такой подход накладывает определённые требования на оформление монадического кода. Так, если до этого функции `A ⇒ B`, которые моделируют вычисление, затрачивающее время, мы представляли как `A ⇒ TimedValue[B]`, для того, чтобы использовать такие функции в наслоении с другими монадами и эффектами, нам придётся представлять из как полиморфные функции по типу эффекта с ограничениями на этот тип эффекта.

Например, функция вычисления квадратного корня вместо того, чтобы быть объявленной как

```
def sqrt(x: Int): TimedValue[Double] =
  math.sqrt(x) |: 1.millisecond
```

должна быть объявлена *полиморфной по типу эффекта*. В стиле *“F“ с дыркой* это будет выглядеть как

```
def sqrt[F[_]: TimedValueAlgebra](x: Int): F[Double] =
  math.sqrt(x) ||: 1.millisecond
```

В стиле монады `Eff`, эта же функция выглядела бы как

```
def sqrt[R: _timedValue](x: Int): Eff[R, Double] =
  math.sqrt(x) ||: 1.millisecond
```

или, используя “стрелочный” синтаксис вместо указания типа `Eff`,

```
def sqrt[R: _timedValue](x: Int): R |▷ Double =
  math.sqrt(x) ||: 1.millisecond
```

Принципиально, в рамках данной работы, оба способа полиморфного задания эквивалентны. Поэтому, для простоты, в этом разделе будем рассматривать только второй способ изложения (при помощи монады `Eff`).

В данном примере типовой параметр `R` отвечает за произвольный эффект, который ограничен тем, что в нём обязан присутствовать эффект `_timedValue`. При этом, `_timedValue` технически объявлен как `TimedValue ⊨ ?`, то есть означает, что эффект, моделируемый монадой `TimedValue` присутствует в стеке монад. Синтаксис оператора `||:` позволяет удобно создавать значения соответствующего типа (то есть, `Eff` с произвольным эффектом `R`, который, в свою очередь, ограничен наличием `TimedValue` в стеке монад).

Перечислим виды значений, с которыми приходится работать в задачах моделирования.

Единичное значение Положим, у нас есть простое целочисленное значение, запакованное в произвольный стек монад (то есть, нет никаких ограничений на R):

```
def simpleVal[R]: R |▷ Int
```

Мы можем применить функцию вычисления квадратного корня к этому значению:

```
simpleVal »= sqrt
```

Надо понимать, что запись выше — сокращённая нотация для частного случая последовательной композиции монадических функций, где результат одной функции подаётся как единственный аргумент следующей. В более общем случае можно использовать нотацию `for – yield`:

```
for {
  v ← simpleVal
  s ← sqrt(v)
} yield s
```

В этой нотации явно присутствуют все промежуточные значения, содержащиеся в монадах, и явно происходит передача аргументов в функции. Эта нотация подходит для случаев, когда используются монадические функции с числом аргументов больше одного и/или при необходимости промежуточных вычислений.

Результатом будет значение типа `R |▷ Double`, где на R будет наложено ограничение наличия `TimedValue` в стеке монад. Мы можем запустить вычисление этого выражения и получим значение типа `Double` на выходе:

```
type S1 = Fx.fx1[TimedValue]

(simpleVal[S1] »= sqrt[S1]).runTimed.run
// результат типа TimedValue[Double]
```

Множественное значение Представим себе, что на входе есть множественное значение, для которого мы хотим выполнить наше timed-вычисление. Такое множественное значение может быть промоделировано следующим образом:

```
def severalVals[R: _list]: R |▷ Int
```

Разница с рассмотренным выше `simpleVal` в том, что эффект R содержит требование множественности `_list` (которое раскрывается в `List ⊨ ?`, наподобие тому, как `_timedValue` раскрывается в `TimedValue ⊨ ?`).

Мы можем таким же образом скомпозировать это множественное значение с функцией вычисления квадратного корня (напомним, что эта функция тратит модельное время):

```
severalVals »= sqrt
```

Мы так же, как и в предыдущий раз, можем запустить вычисление этого выражения. Однако, до момента запуска результирующее значение имеет неоднозначность порядка запуска. Компилятор не знает хотим ли мы получить результат вычисления списка, занимающий модельное время, или же список результатов, каждый занимающий своё модельное время на вычисление. Это определяется

типом результата: в первом случае от `TimedValue[List[Double]]`, во втором он `List[TimedValue[Double]]`. Мощность подхода с использованием стека монад состоит в том, что мы можем получить оба варианта результатов, и при этом отдельные участки кода (в данном случае, функции `severalVals` и `sqrt`) будут одинаковыми в обоих случаях, то есть собственно описание алгоритма вычислений полностью переиспользуется:

```
type S2 = Fx.fx2[TimedValue, List]

(severalVals[S2] »= sqrt[S2]).runList.runTimed.run
// результат типа TimedValue[List[Double]]

(severalVals[S2] »= sqrt[S2]).runTimed.runList.run
// результат типа List[TimedValue[Double]]
```

Разница есть только в порядке вызова интерпретаторов `runList` и `runTimed`.

Вероятностное значение Аналогичным образом мы можем переиспользовать функцию `sqrt` для работы с вероятностным значением. Имея `variadicVal`, возвращающее конечное дискретное распределение целых чисел, мы можем запустить следующий код и получить дискретное распределение чисел `Double`, занимающих модельное время:

```
type S3 = Fx.fx2[TimedValue, DiscreteFiniteDistribution]

(variadicVal[S3] »= sqrt[S3]).runDFD.runTimed.run
// результат типа DiscreteFiniteDistribution[TimedValue[Double]]
```

Более сложные смешения В общем случае мы можем использовать стек произвольной глубины. В качестве примера, мы можем рассмотреть ситуацию, когда timed-вычисление зависит от источников разных монадических функций. Для простоты, мы будем использовать описанные выше функции `severalVals` и `variadicVal`, а в timed-функцию `sqrt` будем передавать сумму от тех двух функций.

```
def f[R: _timedValue:_list:_dFD]: R |▷ Double = for {
  v1 ← severalVals
  v2 ← variadicVal
  s ← sqrt(v1 + v2)
} yield s
```

В общем случае, существует шесть вариантов выполнить эту функцию и получить конкретное значение. Все эти варианты будут давать результат различных типов (и, как следствие, вычисление будет иметь различную семантику). При этом, для получения этих значений будет использоваться один и тот же код функции `f`.

Рассмотрим несколько примеров таких вычислений. В первом случае мы будем получать список вероятностных распределений timed-величин типа `Double`. Во втором случае мы будем получать вероятностное распределение timed-величин типа `List[Double]`.

```
type S4 = Fx.fx3[TimedValue, List, DiscreteFiniteDistribution]

f[S4].runTimed.runDFD.runList.run
// результат типа List[DiscreteFiniteDistribution[TimedValue[Double]]]
```

```
f[S4].runList.runTimed.runDFD.run
// результат типа DiscreteFiniteDistribution[TimedValue[List[Double]]]
```

Таким образом мы можем использовать вычисления, описывающие вычисления, занимающие модельное время, вместе с другого рода эффектами. При этом, легко осуществляется разделение описания действий и выполнения этих действий. Вместе с этим, способ выполнения определяет конечную семантику всей операции, которая задаётся упорядочиванием эффектов из стека монад. Это позволяет переиспользовать один и тот же код поведений в различных ситуациях и даёт возможность не принимать определённых проектных решений (в частности, по упорядочиванию эффектов) раньше времени.

Обобщения затрачиваемого времени

Значение типа `TimedValue` содержит единственное значение времени, обозначающее длительность — модельное время, затрачиваемое для вычисления хранимого значения. В общем же случае может потребоваться иметь не простое значение длительности, а более сложные варианты. В частности, рассмотрим следующие примеры таких потребностей.

Косвенное время Например, при описании вычисления не всегда известно собственно затрачиваемое время на выполнение. Время может зависеть от используемого процессора, затрат на коммуникацию, работы кэша и пр. При детальном моделировании времени, это может существенно сказаться на результате вычислений.

В частности, вместо указания значений затрачиваемого модельного времени, может быть потребность в указании затрачиваемых модельных тиков процессора или в указании числа различных инструкций определённого вида, количества обращений к памяти и подобном.

То есть, в конечном итоге, время указывается *косвенно*, в терминах процессора или аппаратуры.

Можно вводить специализированные аналоги монады `TimedValue` для такого рода косвенного задания времени. Однако, могут возникнуть проблемы, связанные с возможной потребностью в разнообразии такого рода монад.

Также возникают проблемы с композицией этих монад с монадой `TimedValue`, так как для корректной композиции они должны знать конкретные параметры используемых процессоров. Это может приводить к необходимости изменения кода, описывающего вычисления при внешних по отношению к нему изменениях (смене типа процессора).

Не единственное значение времени Так, не всегда может быть известно точное (пусть даже и модельное) время выполнения тех или иных функций. Единственное значение времени, использующееся в монаде `TimedValue` можно, конечно, интерпретировать как ограничение сверху. Правда, в таком случае, мы лишаемся возможности оценивать затрачиваемое модельное время снизу, что может быть нужным иногда [Tr16].

Проблемы с нижней границей можно пробовать решать введением интервала времён вместо отдельного времени в `TimedValue`. Соответственным образом нужно будет изменить правила композиции монады (со сложением интервалов вместо

сложения отдельных величин и соответствующей композицией интервалов при параллельной композиции).

Но интервал может стать лишь грубым приближением действительно необходимого значения времени. Аналогично различным входным параметрам вычислений при построении стека монадических вычислений, может потребоваться выражать, например, конкретный список отдельных времён (а не непрерывный интервал) или даже вероятностное распределение затрачиваемых времён.

Если обобщить эти мысли, получится, что монада `TimedValue` должна содержать не *отдельное единичное значение времени*, а *обобщённое значение, содержащее время*.

```
case class TimedValue[A, F[_]](a: A, time: F[Time])
```

При этом, например, для композиции может требоваться, чтобы данный `F[_]` был, например, функтором. Для параллельной композиции, соответственно, может требоваться, чтобы `F[_]` был бы аппликативным функтором. Аналогично, для последовательной композиции может быть аналогичное требование для монадичности этого параметра типа. Также, могут существовать и другие дополнительные ограничения на `F[_]`.

Подобные ограничения на `F[_]` могут быть отнесены на как можно более поздний момент по коду (то есть, в функциях, которые действительно используют эти свойства (функторности, аппликативности, монадичности или другие).

Можно рассматривать, также, обобщение (с одной стороны усиляющее, с другой более ограничивающее), когда хранимое время — это не обёрнутое значение, а произвольный стек монад, в конечном счёте продуцирующий значение времени (в соответствии с семантикой стека).

```
case class TimedValue[A, R](a: A, time: R |▷ Time)
```

Более того, это обобщение позволяет естественным образом описывать случай, когда время задано косвенно (например, через затрачиваемые тики процессора и пр.). В таком случае, эти данные должны позволять получить конечное время через промежуточные монады, которые в конечном итоге показываются в стеке монад.

```
for {
  a ← timedVal ||: (1 to 5).milliseconds
  b ← tickedVal(a) ||: (100 to 200).processorTicks
  c ← instVal ||: (500.arithOps + 20.controlOps)
} yield b + c
```

Стоит отметить, что даже при использовании непрямого представления времени, стеки монад позволяют задавать интервалы, множественности или даже вероятностные распределения затрачиваемого времени через тики процессора и другие способы. При этом, при смене типа процессора и других характеристик (и, как следствие, возможного изменения затрачиваемого модельного времени), код, описывающий вычисления, не будет меняться.

Контроль за количеством времени на уровне типов

Рассмотрим другой вариант изменения базовой идеи. Мощная система типов позволяет более точно контролировать затрачиваемое модельное время и соответствие имеющегося и требуемого модельного времени.

В частности, есть возможность отразить, что определённый исполнитель timed-функций может принять как аргумент только функции, затрачивающие не более, чем заданное количество времени. Это может быть полезно для статической проверки в случае использования со строго-периодическими операционными системами, которые находят широчайшее применение в ответственных системах реального времени.

Само затрачиваемое модельное время можно присоединить к выражению при помощи литеральных типов, о которых говорилось ранее. Так, например, в следующем коде

```
val x = for {
  a ← 1 ||: 5.milliseconds
  b ← 15 ||: 3.milliseconds
  c ← sumTimed(a, b)
} yield c
```

переменная `x` будет иметь тип, соответствующий результату типа `Int` вычисляемому за 10 миллисекунд модельного времени. Если мы попытаемся передать это выражение в функцию, которая может выполнить выражения, занимающее не более, например, восьми миллисекунд модельного времени, мы получим ошибку времени компиляции.

Для того, чтобы это сделать, нужно, чтобы тип, соответствующий затрачиваемому модельному времени, был приписан к timed-выражению. Это можно сделать как минимум двумя способами.

Типовой параметр Одним из способов является добавление типово́го параметра в монадический тип, описывающий выражение, вычисляемое за модельное время, то есть в тип `TimedValue`.

Таким образом, этот тип начинает иметь сигнатуру не `TimedValue[A]`, а `TimedValue[A, TimeBound]`. Соответственно, все операции над этим типом должны в своих сигнатурах учитывать этот типовой параметр.

Это, на самом деле, представляет собой некую проблему для некоторых методов, потому что они перестают отвечать своим абстракциям. Так, общий вид функции `flatMap` для монад `F[_]` следующий:

```
def flatMap[A, B](a: F[A])(f: A ⇒ F[B]): F[B]
```

Из этого видно, что если для какого-то зафиксированного `TimeBound` (например, `TB1`), мы примем `F[A] = TimedValue[A, TB1]`, из этого будет следовать, что функцию `flatMap` для значения `TimedValue[A, TB1]` можно применить только к значению `TimedValue[B, TB1]` (то есть, с тем же типовым параметром занимаемого времени `TB1`), но и что возвращаемое значение тоже должно иметь второй типовой параметр равным `TB1` (а не сумме `TB1` и `TB1`, как должно быть по семантике последовательной композиции). Это ограничивает применимость монадического подхода.

Однако, существует обобщение монад на случай, когда некоторый параметр производимого монадой эффекта присутствует в типе. Это получило название *монад, параметрических по эффекту* (parametric effect monads) [Kat14] или *индексированных монад* (indexed monads) [Orch14a], [Orch14b]. Это обобщение было придумано как один из вариантов подхода к гранулированному контролю побочных эффектов, но, судя по всему, очень подходит для случая отображения количества затрачиваемого времени в типе.

Тип-поле Другой способ привязать typelevel-значение к выражению со временем — это *зависимые типы*, а в частности несколько ограниченный вариант, который называется *path-dependent type* [DOT14]. Подход представляет собой добавление поля в структуру данных `TimedValue`, которое является *типом*, а не значением.

```
case class TimedValue[A](a: A, time: Time) {
  type TimeBound
}
```

Реализовывать само ограничение по времени можно тоже двумя способами. В обоих случаях типовым параметром является *конструктивное доказательство* (или *свидетельство*, evidence) того, что содержимое значения времени является ограниченным (например, не превышает заданную величину).

Специальный evidence для ограничения времени Специальный тип `LessThan[T]`, являющийся типом, моделирующим для компилятора предикат. Во многом аналогичен подобным типам, использующимся для описания ограничений на уровне типов (type-level programming), например [RefL]. Параметр `T` описывает при помощи литерального типа то значение, меньше которого должно быть значение, к которому применён этот предикат.

Refined-тип для времени В этом случае, используется библиотека refined types [RefT] и тип-ограничение является типовым параметром для специального типа `Refined`. Тогда, `TimedValue[A, TimeBound]` по сути является парой `(A, Time Refined TimeBound)` вместо просто `(A, Time)` как было описано ранее.

Заключение

Монадический подход к моделированию времени фактически является применением хорошо известных техник функционального программирования к области архитектурного моделирования, в частности к области моделирования поведения сложных компьютерных систем.

В частности, строгая типизированность и широкие возможности мощной системы типов позволяют выражать многие аспекты модели системы напрямую в коде, которые будут проверены на стадии компиляции кода, сгенерированного на основе модели системы. Это позволяет обеспечивать ранние проверки композируемости и соответствия отдельных поведений при изменениях модели и при изменениях отдельных спецификаций поведения компонентов модели. Эти проверки производятся статически, до запуска моделирования всей системы.

Также, появляются широкие возможности по композиции разнородных поведений или их аспектов за счёт использования ограниченного параметрического полиморфизма функций. За счёт использования полиморфизма высокого порядка по эффектам, появляется возможность наслаивания дополнительных эффектов к имеющемуся коду без его переписывания (и даже без перекомпиляции). Это позволяет расширять и уточнять семантику отдельных поведений и их композиций без потери корректности.

Помимо прочего, данный подход обладает другими преимуществами функционального программирования, в том числе возможность использования повышенного уровня абстракции в коде (как следствие, увеличения скорости разработки и простоты понимания), широкие возможности по автоматической оптимизации кода во время компиляции

и автоматическому распараллеливанию во время выполнения, а также многие другие возможности. Конечно, при этом стоит учитывать непривычность такого рода подхода для многих профессионалов в области, что необходимо решать в будущем и не относится напрямую к идеям, изложенным в этой работе.

В рамках проекта РФФИ 17-01-00504 в 2019 году планируется разработать экспериментальный комплекс инструментальных средств для моделирования систем управления. Комплекс будет строиться на платформе Scala, имеющей все необходимые нам средства функционального программирования.

Совместное использование механизмов функционального программирования дает возможность описывать вычисления с минимальными требованиями на инфраструктуру такие, что они гарантированно корректно композируются с другими вычислениями в произовольных подходящих инфраструктурах (при этом, их невозможно использовать в неподходящих инфраструктурах, потому что проверки осуществляются на уровне компилятора). Это позволяет переиспользовать (даже без перекомпиляции) имеющийся код не только для решения конкретной задачи, для которой этот код был разработан, но и для решения задач расширяющего класса.

Как следствие, наращивание набора инструментальных средств и видов анализа в перспективной среде моделирования будет требовать лишь добавления или изменения кода, ответственного за конкретные частные аспекты (в частности, специфичные для задачи виды входных данных и анализ специфичных результатов) и не будет затрагивать имеющийся (полиморфный) код. При этом использование мощных механизмов функционального программирования требует определенной культуры и следования определенному стилю программирования, что в контексте разработки и аттестации систем ответственного назначения является уместной платой за получение преимуществ в плане повышения доли повторно используемых компонентов (re-use), композируемости и достоверной консистентности системы.

Литература

Buz14 Denis Buzdalov, Alexey Khoroshilov. A Discrete-Event Simulator for Early Validation of Avionics Systems. In Proceedings of the First International Workshop on Architecture Centric Virtual Integration co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), Valencia, Spain, September 29, 2014

http://ceur-ws.org/Vol-1233/acvi14_submission_3.pdf

Buz16 Denis Buzdalov, Simulation of AADL models with software-in-the-loop execution, ACM SIGAda's High Integrity Language Technology International Workshop on Model-Based Development and Contract-Based Programming as part of Embedded Systems Week (ESWEEK), October 6-7, 2016, Pittsburgh, Pennsylvania, USA

http://sigada.org/conf/hilt2016/paper-Buzdalov.pdf

CPrf Performance characteristics, Scala collections documentation

https://docs.scala-lang.org/overviews/collections/performance-characteristics.html

Car85 Luca Cardelli, Peter Wegner. On understanding types, data abstraction, and polymorphism. ACM Computing Surveys (CSUR) - The MIT Press scientific computation

series Surveys Homepage archive, volume 17, issue 4, 1985, pp. 471-523, doi: 10.1145/6041.6042

http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf

CtsEff The IO Monad for Scala

https://typelevel.org/cats-effect/

CtsPar `Parallel` typeclass, cats library.

https://typelevel.org/cats/typeclasses/parallel.html

DOT14 Nada Amin, Tiark Rompf, Martin Odersky. Foundations of Path-Dependent Types. Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, Portland, Oregon, USA, October 20–24, 2014, pp. 233-249, doi:10.1145/2660193.2660216

http://lampwww.epfl.ch/~amin/dot/fpdt.pdf

Foy17 Julien Richard Foy, Do it with (free?) arrows! Typelevel Summit Copenhagen, June 2017

https://www.youtube.com/watch?v=PWBTOhMemxQ

Hug00 John Hughes, Generalising monads to arrows. Science of Computer Programming, Volume 37, Issues 1–3, May 2000, Pages 67–111, Elsevier, doi:10.1016/S0167-6423(99)00023-4

https://www.sciencedirect.com/science/article/pii/S0167642399000234

Hug04 John Hughes, Programmins with Arrows. In Advanced Functional Programming, 2004, doi:10.1007/11546382_2

http://www.cse.chalmers.se/~rjmh/afp-arrows.pdf

Hug84 John Hughes, “Why functional programming matters”. PMG-40, Chalmers University of Technology, Goteborg, Sweden, 1984

http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf

JDG18 John A De Goes. Blazing Fast, Pure Effects without Monads. LambdaConf conference, Boulder, CO, USA, June 2018

https://www.youtube.com/watch?v=L8AEj6IRNEE

Kat14 Shin-ya Katsumata, Parametric effect monads and semantics of effect systems. POPL ’14 Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 633-645, San Diego, California, USA, January 22–24, 2014, doi:10.1145/2535838.2535846

https://dl.acm.org/citation.cfm?id=2535846

Kis12 Oleg Kiselyov, Typed Tagless Final Interpreters. In: Gibbons J. (eds) Generic and Indexed Programming. Lecture Notes in Computer Science, vol 7470. Springer, Berlin, Heidelberg, 2012, doi:10.1007/978-3-642-32202-0_3

http://okmij.org/ftp/tagless-final/course/lecture.pdf

Kis13 Oleg Kiselyov, Amr Sabry, Cameron Swords. Extensible Effects: An Alternative to Monad Transformers. Proceedings of the 2013 ACM SIGPLAN symposium on Haskell, pp. 59-70. Boston, MA, USA. September 23-24, 2013, doi:10.1145/2503778.2503791

http://okmij.org/ftp/Haskell/extensible/exteff.pdf

Kis15 Oleg Kiselyov, Hiromi Ishii. Freer Monads, More Extensible Effects. Proceedings of the 2015 ACM SIGPLAN symposium on Haskell, pp. 94-105. Vancouver, BC, Canada. September 3-4, 2015, doi:10.1145/2804302.2804319

http://okmij.org/ftp/Haskell/extensible/more.pdf

Kis16 Oleg Kiselyov, Having an Effect. At Institute of Information Science, Academia Sinica, Taipei, Taiwan, December 2, 2016

http://okmij.org/ftp/Computation/having-effect.html

https://www.youtube.com/watch?v=GhERMBT7u4w

Leo14 George Leontiev, There's a prolog in your scala. ScalaIO conference, Paris, France, October 2014

https://www.youtube.com/watch?v=iYCR2wzfdUs

MTL95 Sheng Liang, Paul Hudak, Mark Jonest. Monad Transformers and Modular Interpreters. POPL '95 Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 333-343, doi:10.1145/199448.199528

https://dl.acm.org/citation.cfm?id=199528

McBr08 Conor McBride, Ross Paterson. Applicative programming with effects. In Journal of Functional Programming, Volume 18, Issue 1, January 2008, pp. 1-13. doi:10.1017/S0956796807006326

http://strictlypositive.org/IdiomLite.pdf

Mog91 Eugenio Moggi, Notions of computation and monads. In Information and Computation, Volume 93, Issue 1, July 1991, Pages 55-92, Elsevier

doi:10.1016/0890-5401(91)90052-4

https://core.ac.uk/download/pdf/82259574.pdf

MxTsk Monix library, Task monad

https://monix.io/docs/3x/eval/task.html

Nizh18 Олег Нижников, Современное ФП с Tagless Final. Конференция Joker 2018, Санкт-Петербург.

https://www.youtube.com/watch?v=sWEtnq0ReZA

Oka96 Chris Okasaki, Purely Functional Data Structures. PhD thesis, School of Computer Science, Carnegie Mellon University

https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf

Oka98 Chris Okasaki, Purely Functional Data Structures. Cambridge University Press, doi:10.1017/CBO9780511530104

Orch14a Dominic Orchard, Tomas Petricek, Alan Mycroft. The semantic marriage of monads and effects. Published at arXiv:1401.5391, 21 Jan 2014

https://arxiv.org/pdf/1401.5391.pdf

Orch14b Dominic Orchard, Tomas Petricek. Embedding effect systems in Haskell. Haskell Symposium, 2014

http://tomasp.net/academic/papers/haskell-effects/haskell-effects.pdf

Pol15 Yuriy Polyulya, Functional programming with arrows. Scala Days conference, 2015, Amsterdam

https://www.youtube.com/watch?v=ZfAgvAIoUEY

RefL `Less`, predicate that checks if a numeric value is less than `N`. refined library

https://github.com/fthomas/refined/blob/master/modules/core/shared/src/main/scala/eu/timepit/refined/numeric.scala#L36

RefT refined, a Scala library for refining types with type-level predicates which constrain the set of values described by the refined type.

https://github.com/fthomas/refined

Run18 Rúnar Bjarnason, Introduction to the Unison programming language. Lambda World 2018 conference, Living Computer Museum, Seattle, 18 september 2018

https://www.youtube.com/watch?v=rp_Eild1aq8

SPJ86 Simon Peyton Jones, Functional programming languages as a software engineering tool. In Embedded Systems. Lecture Notes in Computer Science, vol 284. Springer, Berlin, Heidelberg, pp 153-173, 1986, doi:10.1007/BFb0016351

https://link.springer.com/content/pdf/10.1007%2FBFb0016351.pdf

SPJ92 Simon Peyton Jones, Philip Wadler. Imperative functional programming. POPL '93 Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 71-84. doi:10.1145/158511.158524

https://www.microsoft.com/en-us/research/wp-content/uploads/1993/01/imperative.pdf

Ste94 Guy L. Steele Jr. Building interpreters by composing monads. In POPL '94, pages 472–492. ACM Press, 1994. doi:10.1145/174675.178068

https://dl.acm.org/citation.cfm?id=178068

Tr16 A.M. Troitskiy, D.V. Buzdalov, A static approach to estimation of execution time of components in AADL models. Proceedings of the Institute for System Programming, vol. 28, issue 2, 2016, pp. 157-172, doi: 10.15514/ISPRAS-2016-28(2)-10

http://ispras.ru/proceedings/docs/2016/28/2/isp_28_2016_2_157.pdf

Wdl11 Sam Lindley, Philip Wadler, Jeremy Yallop. Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous. Electronic Notes in Theoretical Computer Science, Volume 229, Issue 5, 8 March 2011, Pages 97–117, Elsevier, doi:10.1016/j.entcs.2011.02.018

https://www.sciencedirect.com/science/article/pii/S1571066111000557

Wdl92 Philip Wadler, Comprehending Monads. In Mathematical Structures in Computer Science, vol 2, issue 4, Cambridge University Press, 1992, pp. 61–78, doi:10.1.1.33.5381 or doi:10.1017/S0960129500001560

https://ncatlab.org/nlab/files/WadlerMonads.pdf

Wel17 Noel Welsh, Uniting Church and State: FP and OO Together. Scala Days conference, Copenhagen, 2017

https://www.youtube.com/watch?v=IO5MD62dQbI

ZIO ZIO, Scala-library for asynchronous and concurrent programming, IO monad

https://scalaz.github.io/scalaz-zio/datatypes/io.html