[Talk] Programming sucks

Sunday, May 11, 2014 2:13 PM

В последнее время я начал всё больше и больше осознавать, что с программированием что-то не так.

Взять, к примеру, недавний релиз Java 8. Я вообще не представляю, как Java-разработчики до сих пор используют этот язык и говорят при этом, что жизнь не такая уж и плохая, да и вообще он приятный. Мне кажется, что типичный день Java-разработчика выглядит вот так [grumpy cat]. Стоит отдать должное этим ребятам, они молодцы, что держатся и всё ещё живут среди нас. Но всё-таки советую встать на путь спасения и попробовать Scala.

Так вот, после долгих лет, под давлением общественности ребята сделали поддержку лямбда-функций в языке. Рассмотрим простой пример - найдём сумму положительных элементов матрицы, представленной в виде списка списков чисел. Вроде бы всё очень даже мило, и мы можем использовать стрелочки, но давайте взглянем на типы "функций" (на самом деле так называемых функциональных интерфейсов). Как нам хотелось бы? Какой-то общий супер-тип для всех функций, аналог тайп-конструктора, насыщая который конкретными параметрами, мы бы получали специфичные типы функций. Звучит достаточно разумно, как в общем-то и сделано во многих современных статически-типизированных языках. Как получилось?

Разработчики JDK посчитали, что вместо унификации лучше создать функциональный интерфейс на каждый возможный сценарий использования лямбда-функций.

В этом есть определённый плюс, теперь можно играть в мини-игры на сообразительность. Например:

Use case	C#	Java
act	Action<T> (Func<T, Unit>	Consumer<T>, DoubleConsumer, IntConsumer, …
	Action<T1, T2>	BiConsumer<T1, T2>
	Action<T1, T2, T3>	DIY
get	Func<T>	Supplier<T>
map	Func<T, R>	Function<T, R>, ToDoubleFunction<T>, ToIntFunction<T>, …
filter	Func<T, bool>	Predicate<T>
compare	Func<T, T, int>	Comparator<T>
reduce	Func<T, T, T>	BinaryOperator<T>
reduce	Func<U, T, U>	BiFunction<U, T, U>

Можно спросить "ну и что, нормальные же вроде бы функции". На самом деле нет. Две причины - отличия от большинства других языков. Унификация это почти всегда хорошо. Вторая - несовместимость библиотек.

Представьте, что у вас есть две независимые библиотеки. В одной из них есть методы, принимающие функции трёх параметров, а в другой есть методы, предоставляющие эти функции. Если каждая библиотека объявляет свой собственный тип, описывающий такие функции, то вам, как разработчику, склеивающему эти библиотеки придётся писать функции-адаптеры, транслирующие функции одной библиотеки в функции другой.

Я начал думать, почему так происходит?
Больше похоже на то, что разработчики JDK решили совсем не утруждать себя анализом аналогичных решений в других языках и платформах и попросту создали десяток функциональных интерфейсов. Чем плох зоопарк функциональных интерфейсов? Основная проблема - совместимость различных библиотек.

Мы не учимся на чужих ошибках. Если бы разработчики JDK проанализировали решения, существующие в других языках и платформах, посмотрели бы на проблемы, с которым столкнулся C# 3.0, то смогли бы сразу же выпустить удобное для использования и расширения решение.

Рассмотрим другой пример. В предпоследней версии C# появилась поддержка операторов async/await, призванных упростить работу с асинхронными вычислениями. Несмотря на то, что в других языках похожие решения существовали уже достаточно давно (call-cc в лиспе, родоначальник всех control-flow извращений?, async builders в F#), релиз этой языковой фичи вызвал достаточно большой резонанс и очень сильное восхищение.

Для тех, кто не знает или не помнит, как работают операторы async/await, вот простой пример из жизни разработчиков, в доменной области, понятной для них. Есть асинхронный процесс, описывающий жизнь определённого разработчика. Он дожидается зарплаты (у нас она вот уже вчера была, но всё-таки), потом эту зарплату пропивает, тоже асинхронно, т.к. это длительный и затяжной процесс, а потом, когда зарплата заканчивается, переходит в новое состояние разочарования и депрессии. Основное назначение await - избавить нас от лапши из коллбеков, вывернув асинхронную операцию и представив её в императивном синхронном виде. Ключевыми компонентами здесь является тип Task, представляющий собой значение (нашего состояния), которое будет когда-то в будущем, а также оператор await, который говорит "дождись получения значения моего операнда", а потом выполни всё то, что находится дальше по коду и верни мне task, который будет завершён когда все последующие операции завершатся.

Очень упрощённо можно представить, что компилятор разворачивает это всё в цепочку коллбеков, возвращающую финальный таск.

К сожалению, абстракция "асинхронные операции возвращают таски" немного протекла. async void - это такая штука, которая у нас есть в C#, но использовать которую нам очень-очень не рекомендуют за исключением одного конкретного случая. И этот случай - асинхронные обработчики событий. Тип события определяет сигнатуру функции, которую можно использовать в качестве обработчика. Чаще всего эти события используются в графических интерфейсах - когда пользователь нажимает на кнопку, нам нужно выполнить какую-то асинхронную операцию. Поэтому дизайнеры языка пошли на компромисс - они разрешили использовать оператор await в методах, которые ничего не возвращают. async void-методы достаточно сильно отличаются от методов, возвращающих таски - они по-другому обрабатывают исключения и подталкивают разработчиков писать асинхронный код без точек синхронизации (в стиле fire-and-forget). Практически в каждом туториале по асинхронности вы встретите фразу "не используйте async void для чего-либо, отличного от обработки событий". Вопрос - почему бы тогда не сделать контракт асинхронных методов более строгим и не заставить компилятор форсить такое правило? Разрешить только методы, возвращающие Task и сделать префикс для подписки на событие асинхронным обработчиком. Пусть компилятор страдает и разворачивает эту в нужные делегаты, отбрасывает результат метода и т.д. Тогда не пришлось бы вставлять костыли в язык и потом писать хитрые гайдлайны по правильному его использованию.

В этом есть большое искусство - делать вещи такими, чтобы их было очень сложно использовать

неправильно.
Другой пример. Напомню, что async/await это языковые конструкции для работы с асинхронными операциями. Как вы думаете, что будет результатом исполнения этого кода?

```
private void Test()
{
    var task = GetTextAsync();
    var text = task.Result;

    MessageBox.Show(text);
}

private async Task<string> GetTextAsync()
{
    await Task.Delay(500);
    return "Hello";
}
```

Правильный ответ - дедлок. Как бы смешно это не было, но с помощью такого, казалось бы, невинного использования языковых конструкций, специально созданных для работы с асинхронным кодом, действительно можно вызвать дедлок. Причина крайне проста - протекание абстракций на стыке синхронного и асинхронного миров. Мы должны знать, что на самом деле асинхронные коллбеки исполняются на специальном диспетчере, который занимается распределением этих коллбеков на конкретные треды. По умолчанию этот диспетчер пытается восстановить оригинальный поток, на которым мы находились, когда инициировали операцию GetTextAsync(). Но он не может этого сделать, т.к. этот поток уже находится в блокированном ожидающем состоянии на вызове .Result. Хотя кажется, что использование очень наивное и безопасное.

Что делать в такой ситуации? Конечно же, воспользоваться волшебным флажком! ConfigureAwait(false) говорит диспетчеру не восстанавливать оригинальный контекст исполнения, что выполнит коллбек на другом потоке. Эта опция, кстати говоря, также рекомендуется к использованию вообще всегда, когда явно не нужно возвращаться на оригинальный поток, для производительности. Этот факт заставляет задуматься - а насколько вообще хорош оператор await, который требует постоянного использования ConfigureAwait(false)?
Ещё один пример с await. У нас есть очень странная синтетическая задача - создать список случайных величин, где каждая случайная величина получается асинхронно. Потом мы решаем, что т.к. эта операция длительная, то неплохо было бы уметь отменять её. В .NET это делается при помощи специального cancellation token - маркера, у которого можно запросить состояние "отменённости" и собственно изменить это состояние. Как выглядит код с поддержкой отмены? Как только мы решаем добавить отмену в одном месте, она начинает расползаться по всему окружающему её коду. Мы должны уметь прокидывать токен отмены с точки входа, инициирующей асинхронную операцию, вниз до каждой мелкой подоперации. Стоит нам где-то забыть это сделать - логика отмены для всей большой асинхронной операции сломается и перестанет работать. Ну а если вы используете стороннюю библиотеку, и в ней нет поддержки отмены, ты вы вообще обречены. Вывод - отмена в C# очень хрупкая. Уже даже R# научился подсказывать, что в коде есть вызовы асинхронных методов, умеющих принимать токены, но не принимающие их.

Очевидно, что нам так не очень хочется. Какое решение? Давайте снова заставим страдать компилятор! Вот пример на F#, который делает всё то же самое, но в нём мы не видим ни одного упоминания cancellation token-ов. Магия? Вовсе нет. Всё благодаря операторам do! и let!, которые в контексте асинхронных вычислений самостоятельно проверяют состояние токена отмены этого вычисления. Т.е. мы переносим хрупкую логику на инфраструктуру, в которой мы работаем, что помогает нам сконцентрироваться на основной логике и избежать глупых недочётов.

К сожалению, мы в общем не умеем строить безопасный дизайн, спасающий нас от ошибок. Решения, которые мы строим, часто получаются достаточно хрупкими, заточенными под конкретный способ использования. Стоит забыть где-то передать параметр или вызвать функции в нужном порядке, как всё рассыпается прямо у нас на глазах. Идеальный дизайн попросту не позволяет использовать решение неправильно.

Хорошим примером правильного ограничивающего дизайна являются монады. Через ограничения они заставляют разработчика использовать определённые концепции безопасным и правильным образом. STM не позволяет вылезти разделяемому ресурсу за пределы монады. IO заставляет держать сайд-эффекты внутри контроллируемой монады, отделяя их от чистых функций.

К сожалению, с монадами есть совершенно другая проблема. И эта проблема заключается в том, как мы учим и учимся. Обучение в программировании - отстой. В контексте познания монад даже есть специальный термин monad tutorial fallacy, хорошо описывающий ситуацию вокруг людей, знакомящихся с функциональным программированием.

Что такое монада? Если спросить двух разных людей - Эрика Мейера, знатного любителя типов, монад и дуальности, то вероятно мы услышим что-то вроде "монада - это моноид в категории эндофункторов", ну или что монада это всего лишь моноид в категории эндофункторов. Или что это тройка из тайп-конструктора, операции return :: a -> M a, поднимающей значение в монаду, и операции bind с сигнатурой M a -> (a -> M b) -> M b, которая маппит внутреннее значение монады, сохраняя её контекст и сайд-эффекты. Ну или же следуя более математическому определению, это операторы unit, эквивалентный return, fmap, поднимающий проекцию обычных значений в проекцию монадических значений и join, схлопывающий монаду монады в монаду значения. Ну и очевидно же, что bind m f = join (fmap f m). Всё просто, верно?

Когда на неподготовленного человека обрушивается такое количество информации, то естественно ожидать от него какой-то такой реакции.

Потом проходит какое-то время, человек начинает разбираться, читать более доступные источники информации, наступает момент, когда он думает, что понимает. Тогда у него происходит осознание - а ведь монады это котики в коробках! Так называемые котомонады. Котиков можно доставать из коробки, наряжать в разные одёжки, класть обратно, или же подменять самих котиков, оставляя сайд-эффекты из жизнедеятельности в коробке. При этом коробка - это не коробка, а контекст вычисления с контролируемыми сайд-эффектами!

Но мы же не учимся так программированию, верно? Как мы изучаем циклы? Разве мы осознаём, что цикл for - это императивная реализация функциональной развёртки? (Объяснение того, что такое unfold, его отношение к циклам).

Почему? Казалось бы, ведь клёво. Можно умничать в интернете. Человек обучается на аналогиях и симметриях. Когда мы постоянно видим и применяем одни и те же концепции, мы строим мысленные связи между ними, учимся со временем строить абстракции на основании мелких конкретных кирпичиков.

Скорее всего, если вы были такими же неудачниками, как и я, то вам довелось заниматься программированием в программе "чертёжник". К примеру, если поставить ученика перед задачей нарисовать 3 горизонтальные линии, то он решит её с использованием парадигмы COPY-PASTE: нарисовать одну линию, затем скопировать код, поменять параметры функции в_точку(). А если надо нарисовать 20 линий, при этом немного отличающихся друг от друга? Отличный сценарий, чтобы осознать, что можно избавиться от повторяющегося кода, если выполнять кусочки

программы с разными параметрами.

Почему бы не делать то же самое и с монадами?

Monad explanation here

Почему страдает обучение? Мне кажется, что потому что мы не интересуемся историей развития отрасли, мы принимаем очень многое как должное, не задумываясь о том, как и почему появлялись конкретные концепции и подходы, и такое игнорирование наследия мешает нам полноценно создавать что-то новое, т.к. мы тратим время на переизобретание старых концепций.

В определённые моменты я ощущаю это на себе. К примеру, раньше я часто путал ковариантность и контрвариантность интерфейсов и делегатов в C#. Для тех, кто не помнит или не знает, что это - если у меня есть классы Person и Developer, то я могу использовать Ienumerable<Developer> как IEnumerable<Person> и Action<Person> как Action<Developer>. Я путался в названиях этих двух концепций, так как не видел в них никакой логики, особенно в связке с модификаторами in/out. Потом я заинтересовался теорией категорий, и даже самые её базовые концепции расставили всё на свои места. Несмотря на то, что вся суть теории категории в максимальной абстракции от чего-бы не было, она помогает начать видео подобное в подобных вещах. Она помогает организованно смотреть на группы подобных друг другу вещей

Базовые концепции крайне просты. Категория - это множество произвольных объектов и множество отношений (стрелок, морфизмов) между ними. Identity, некоторые законы - ассоциативность, композиция. Под это описание можно подвести практически что угодно - множество людей и родственные связи между ними, типы стандартной библиотеки и отношение наследования. К категории применимы функторы - действия, сопоставляющие одной категории другую, при этом переносящие как объекты, так и их морфизмы. Если функтор не меняет направление морфизмов, то такой функтор называется ковариантным, иначе - контрвариантным. <Пример с типами и функторами Action, Func>.

Для категорий можно определить огромное количество различных "свойств" - некоторых правил, которые выполняются в терминах данной категории. Одним из наиболее интересных свойств является дуальность.

Дуальность?
Понимание дуальности даёт какое-то необъяснимое чувство гармонии. Ты начинаешь понимать, что практически все используемые нами в работе идеи как-то связаны между собой. Ты учишься лучше выделять абстракции (*лол, выделять абстракции!*) в повседневной работе.

К сожалению, изучение абстрактных концепций ведёт к необратимому разрушению мозга.
Теория категорий слишком абстрактна. Иногда настолько, что попытки построения абстракций не приводят ни к каким практическим результатам. Однако как только вы ступите на тропу формализации и абстрагирования, ваш внутренний перфекционизм начнёт болезненно опухать, доставляя страдания от несовершенства окружающего мира. Вы начнёте осознавать, что разработчики языков и библиотек тоже делают ужасные, зачастую крайне иррациональные ошибки. Вы будете долго страдать, скатываясь всё ниже и ниже (дальше слайды с grumpy cat). *Почему разработчики языков не продумывают абстракции? Почему оператор сложения в C# является функцией, но его нельзя использовать как функцию высшего порядка? Почему то же самое нельзя делать с конструкторами? Почему мы не можем использовать тайп-параметры, параметризованные другими тайп-параметрами?* К тому же, чем ниже вы будете опускаться, тем больше будете осознавать, как много необъятной информации вокруг, как много на самом деле вы ещё не знаете. А когда вы будете на самом дне, вы поймёте, что на самом деле проблема не в окружающем мире, а в вас самих. … Не изучайте теорию категорий!