sheknitrtch 19 апреля 2012 в 11:45

Реклама

Функциональное программирование для вс

Автор оригинала: Slava Akhmechet

Программирование

Перевод

Доброго времени суток. Это статья — перевод заинтересовавш
Университета штата Нью-Йорк в Стоуни-Брук. Статья в доступ
концепции функционального программирования, их преимуще
полезна широкому кругу читателей, которые сомневаются, ну
функционального программирования или нет. Пожелания, пре
терминологии принимаются по личной почте.

*Мнение переводчика может иногда не совпадать с мнением ав
крайне занимательно.*

UPD: альтернативный вариант перевода вы можете найти на rsdn (спасибо @flamingo

Понедельник, 19 июня, 2006

Введение

Программисты — прокрастинаторы (то есть лентяи). Прийти, сделать кофе, проверить почту, почитать RSS летну, почитать новости, проверить свежие статьи на техническом сайте, полистать политические дискуссии на программистском форуме. Смыть, повторить, чтобы ничего не пропустить. Пойти на обед. Вернуться, уткнутся в IDE на несколько минут. Проверить почту. Приготовить кофе. И вот неожиданно день подходит к концу.

Но время от времени в поле зрения оказываются занятные (затруднительные, многообещающие) статьи и посты в блогах. Если вы ищете в правильном месте, то как минимум одна такая статья будет встречаться вам каждые несколько дней. В этих постах сложно разобраться, на это требуется время, поэтому они накапливаются в папке «Прочитать». Прежде, чем вы успеете понять что к чему, окажется что уже накопилась куча ссылок и папка, полная PDF файлов. Вам, и понадобится целый год и хижина посреди леса, где на мили вокруг нет ни души, чтобы во всём этом разобраться. И желательно, чтобы кто-нибудь каждое утро приносил еду и забирал мусор, пока вы прогуливаетесь к реке.

Я не знаю, какой у вас список, но большая часть моего списка касается функционального программирования. Такие статьи обычно самые сложные. Часто они написаны сухим академическим языком, и даже «ветераны Wall Street с десятилетним стажем» не понимают о чем говорится в статьях по функциональному программированию (ФП). Если вы зададите вопрос менеджеру проекта в Citi Group или в Deutsche Bank [1] почему они выбрали JMS вместо Erlang, то услышите в ответ, что они не могут использовать академический язык для промышленных разработок. Проблема в том, что большинство комплексных систем с самыми жесткими требованиями были написаны с использованием элементов функционального программирования. Что-то не сходится.

Действительно статьи о ФП трудны для понимания, но их можно написать проще. Причина возникшей пропасти знаний чисто историческая. По сути в концепции ФП нет ничего сложного. Взгляните на эту статью, как на «доступное руководство по ФП», как на мостик между нашими императивными умами и миром ФП. Заварите себе кофе и продолжим. Надеюсь, что уже очень скоро коллеги начнут шутить по поводу ваших ФП комментариев.

Так что же такое ФП? Откуда оно пошло? На что оно годно? Если оно действительно так полезно, как об этом твердят защитники ФП, то почему его не используют чаще в промышленных масштабах? Почему только люди с PhD склоняются к функциональным языкам? Но что ещё важнее, почему так чертовски трудно освоить функциональные языки? Что скрывается за всеми этими замыканиями, продолжениями, каррированием, ленивые вычисления и отсутствие побочных эффектов? Как это всё можно использовать в проекте, который не охватывает целую вселенную? Почему это всё так далеко от того хорошего, что свято и дорого нашим императивным сердцам? Скоро мы во всём разберёмся. Для начала давайте поймём в чём причина огромной пропасти между академическими статьями и реальным миром. Чтобы ответить на этот

вопрос достаточно прогуляться в парке.

Прогулка в парке

Садитесь в машину времени. Наша прогулка в парке произошла более 2 тысяч лет назад в один из тех солнечных дней давно забытой весны 380 года до н.э. За городскими стенами Афин под ласковыми тенями оливковых деревьев Платон прогуливался в направлении Академии с красивым мальчиком рабом. Стояла чудесная погода, обед приятной тяжестью отдавался в животе, и разговор плавно перешёл на философские темы.

«Посмотри на тех двух студентов», сказал Платон, аккуратно подбирая слова, чтобы придать вопросу образовательного смысла. «Кто из них, по-твоему, выше?» Мальчик раб посмотрел в сторону бассейна, возле которого стояли два человека. «Они приблизительно одного роста», ответил мальчик. «Что ты имеешь в виду под словами 'приблизительно одного роста'?», спросил Платон. «Ну, отсюда они выглядят одинаково, но я уверен, что если мы подойдём поближе, то я смогу увидеть разницу в росте.»

Платон улыбнулся. Он вёл мальчика в правильном направлении. «То есть ты хочешь сказать, что в мире нет идеально совпадающих вещей?» Мальчик призадумался и ответил: «Да, я так думаю. Всегда существует маленькая разница, даже если мы не можем её увидеть.» Он дошёл до самой сути! «Тогда если нет идеально совпадающих вещей в этом мире, то как ты понимаешь концепцию 'идеального' равенства?» Это ввело мальчика в ступор. Он ответил: «Я не знаю».

Так родилась первая попытка понять природу математики. Платон предположил, что в нашем мире всё лишь приближение идеала. Он также осознал, что люди способны понять концепцию идеала, хотя никогда с ним не сталкивались. Он пришёл к выводу, что идеальные математические формы должны существовать в другом мире, и что мы каким-то образом знаем о них из связей с этой «альтернативной» вселенной. Очевидно, что мы не можем увидеть идеальный круг. Но при этом мы понимаем, что из себя представляет идеальный круг, и как он может быть описан математически. Что же тогда математика? Почему вселенная описывается математическими законами? Всё ли может описать математика? [2]

Философия математики очень сложный предмет. Как и большинство философских дисциплин, она скорее задаёт вопросы, чем отвечает на них. Учёные в большинстве своём согласны с тем фактом, что математика — это настоящая головоломка: в основе лежит набор базовых непротиворечивых принципов и набор правил, как этими принципами оперировать. Затем мы можем комбинировать правила, получая всё более и более сложные законы. Математики называют такой метод «формальной системой» или «исчислением». Например, можно построить формальную систему для Тетриса. По сути работающий Тетрис и есть сам по себе формальная система, просто она записана в необычном виде.

Цивилизация пушистых существ с Альфы Центавра не сможет прочесть нашу формальную систему Тетриса или круга, потому что их единственный орган чувств может воспринимать только запахи. Возможно они никогда не построят Тетрис, но наверняка у них будет формальная система для круга. Скорее всего у нас не получится с ней ознакомиться, так как наше обоняние не настолько развито. Но как только будет расшифрован язык представления (путём различных сенсорных инструментов и стандартной техники обратного инженеринга), базовые концепции станут понятны любой интеллектуально развитой цивилизации.

Даже если бы не существовало ни одной разумной цивилизации во вселенной, формальная система для Тетриса и круга всё равно были бы логически верными. Просто не нашлось бы существ, способных эти системы найти и формализовать. Если внезапно появится разумная расса пришельцев, то они, скорее всего, разработают свою формальную систему для описания вселенной. Конечно, маловероятно, что они изобретут Тетрис, потому что во вселенной нет аналогов этой игре. Тетрис — это один пример из огромного числа формальных систем, загадок, которые не имеют отношения к окружающей действительности. Даже такое понятие как натуральные числа не всегда можно отнести к реальному миру, ведь можно представить себе настолько большое число, что его нельзя применить к чему-либо во вселенной, но при этом оно будет конечным.

Немного истории [3]

Давайте повернём колёса нашей машины времени и переместимся немного ближе, в 1930-е. Великая депрессия опустошила Новый и Старый свет. Почти все семьи из всех социальных слоев почувствовали на себе громадный экономический спад. Осталось совсем мало убежищ, в которых люди могли не боятся бедности. Некоторым людям повезло оказаться в таких убежищах. Нас интересуют математики в Принстонском университете.

Новые корпуса, построенные в готическом стиле, придавали университету ауру безопасности. Специалисты по логике со всей страны приглашались в Принстон для основания нового подразделения. В то время, как большинство американцев с трудом добывали себе пропитание, высокие потолки, стены с узорами из дерева, ежедневные дискуссии за чашечкой чая, и прогулки в лесу, составляли условия проживания в Принстоне.

Одним из математиков, проживавших в таком расточительном образе жизни, был молодой человек по имени Алонзо Чёрч (Alonzo Church). Алонзо получил степень бакалавра в Принстоне и его уговорили остаться в аспирантуре. Алонзо чувствовал, что окружающая обстановка была чересчур роскошной. Он редко появлялся на обсуждении математических проблем за чашечкой чая и не любил гулять в лесу. Алонзо был одиночкой: он был более плодовит, когда работал один. Тем не менее он регулярно встречался с другими обитателями Принстона. Среди которых были Алан Тьюринг (Alan Turing), Джон фон Нейман (John von Neumann) и Курт Гёдель (Kurt Gödel).

Эти четверо интересовались формальными системами. Они не уделяли особого внимания физическому миру, их интересовала работа с абстрактными математическими головоломками. В их головоломках было нечто общее: математики изучали вопросы вычислений. Если у нас есть машина с бесконечными вычислительными возможностями, то какие задачи можно на ней решать? Можно ли решать задачи автоматически? Существуют ли неразрешимые задачи и почему? Будут ли машины с разной архитектурой одинаковыми по мощности?

Совместно с другими учёными Алонзо разработал формальную систему названную Лямбда-исчислением. Система по сути была языком программирования для одной из воображаемых машин. Она была основана на функциях, которые принимают в качестве аргументов функции, и возвращают функцию. Такая функция была обозначена греческой буквой Лямбда, что дало название всей системе [4]. Используя эту систему Алонзо удалось построить рассуждения касательно вышеописанных вопросов и вывести ответы на них.

Независимо от Алонзо, Алан Тьюринг проводил подобное исследование. Он разработал другую формальную систему (которую сейчас называют Машиной Тьюринга), и используя её пришёл к выводам, подобным Алонзо. Позже было доказано, что машина Тьюринга и лябда-исчисление имеют одинаковую мощность.

В этот момент наша история останавливается. Я бы подытожил статью, и вы переключились бы на другую страницу, если бы не началась Вторая мировая война. Мир пылал. Войска США очень интенсивно использовали артиллерию. Чтобы повысить точность армия наняла большую группу математиков, которые постоянно решали дифференциальные уравнения, необходимые для баллистических таблиц стрельбы. Быстро стало понятно, что такая задача слишком сложна для ручного решения, для преодоления этой проблемы было разработано специальное оборудование. Первой машиной для решения баллистических таблиц был Mark I построенный IBM — она весила 5 тонн, состояла из 750'000 деталей и могла совершать 3 операции в секунду.

Гонка, конечно, на этом не закончилась. В 1949 общественности был показан Электронный Дискретный Переменный Автоматический Компьютер (Electronic Discrete Variable Automatic Computer, EDVAC). Это был первый пример реализации архитектуры фон Неймана, и был первой действительно работающей машиной Тьюринга. На некоторое время работы Алонзо Чёрча были отложены в сторонку.

В конце 50-ых профессор Массачусетского технологического института (MIT) Джон Маккарти (John McCarthy), тоже выпускник Принстона, начал проявлять интерес к работе Алонзо Чёрча. В 1958 году он представил язык обработки списков, **Lis**t **P**rocessing language (Lisp). Lisp задумывался как имплементация Лямбда-исчисления Алонзо, которая работает на компьютерах фон Неймана. Многие компьютерные учёные отметили выразительную мощь Lisp-а. В 1973 году группа программистов в лаборатории искусственного интеллекта в Массачусетском технологическом институте разработали железо, которое они назвали Lisp-машиной. Это была аппаратная реализация лямбда-исчислений Алонзо.

Функциональное программирование

Функциональное программирование — это практическая реализация идей Алонзо Чёрча. Не все идеи Лямбда-исчисления переросли в практическую сферу, так как лямбда-исчисления не учитывали физических ограничений. Тем не менее, как и ОО программирование, функциональное программирование — это набор идей, а не набор четких указаний. Существует много функциональных языков, и большинство из них делают одни схожие вещи по разному. В данной статье я объясню наиболее широко используемые идеи из функциональных языков используя примеры на Java (да, вы можете писать функциональные программы на Java если у вас есть склонности к мазохизму). В следующих нескольких разделах мы возьмём язык Java и внесём в него изменения, чтобы он превратился в пригодный к использованию функциональный язык. Начнём наше путешествие.

Лямбда исчисление было придумано для изучения проблем, связанным с вычислениями. Функциональное программирование, стало быть, в первую очередь имеет дело с вычислениями, и, на удивление, использует для этого функции. Функция — это базовый элемент функционального программирования. Функции используются почти для всего, даже для простейших расчётов. Даже переменные заменяются функциями. В функциональном программировании переменные — это просто синонимы (alias) для выражений (чтобы нам не пришлось писать всё в одну строку). Их нельзя изменять. В каждую переменную можно записать только один раз. В терминах Java это означает, что все переменные

объявляются как `final` (или `const` если имеем дело с C++). В ФП нет не-`final` переменных

```
final int i = 5;
final int j = i + 3;
```

Так как все переменные финальные, то можно сформулировать два утверждения. Нет смысла постоянно писать ключевое слово `final`, и нет смысла называть переменные … переменными. Теперь мы внесём два изменения в Java: каждое объявление переменной будет финальным, мы будем обращаться к переменным как к символам.

Теперь вы, наверное, удивляетесь, как вообще можно написать что-либо достаточно сложное на таком языке. Если все символы неизменяемые, то мы в принципе не можем поменять состояние программы! Это не совсем верно. Когда Алонзо работал над лямбда-исчислением, у него не было нужды сохранять состояние, чтобы изменить его позже. Его интересовало проведение операций над данными. Тем не менее, было доказано, что лямбда исчисление эквивалентно машине Тьюринга. В нём можно делать всё то же, что возможно в императивных языках. Как же нам достичь тех же результатов?

Оказывается, что функциональные программы могут хранить состояние, только они не используют для этого переменные. Они используют функции. Состояние хранится в параметрах функции, в стеке. Если хотите сохранить состояние, чтобы потом изменить его через время, то нужно написать рекурсивную функцию. Например, давайте напишем программу, которая переворачивает Java строку. Не забудьте, что все переменные объявляются как `final` [5].

```
String reverse(String arg) {
    if(arg.length == 0) {
        return arg;
    }
    else {
        return reverse(arg.substring(1, arg.length)) + arg.substring(0, 1);
    }
}
```

Эта функция довольно медленная, потому что она повторно вызывает сама себя [6]. Здесь возможна утечка памяти, так как множество раз создаются временные объекты. Но это функциональный стиль. Вам может показать странным, как люди могут так программировать. Ну, я как раз собирался вам рассказать.

Преимущества функционального программирования

Вы, наверное, думаете, что я не смогу привести доводы в оправдание монструозной функции выше. Когда я только начинал изучать функциональное программирование, я тоже так думал. Я ошибался. Есть очень хорошие аргументы в пользу такого стиля. Некоторые из них субъективные. Например, программисты заявляют, что функциональные программы проще понять. Я не буду приводить таких аргументов, потому что всем известно, что лёгкость понимания — это очень субъективная вещь. К счастью для меня, есть ещё куча объективных аргументов.

Unit тестирование

Так как в ФП каждый символ является неизменяемым, то функции не имеют побочных действий. Вы не можете менять значения переменных, к тому же функция не может поменять значение вне своей области видимости, и тем самым повлиять на другие функции (как это может случится с полями класса или глобальными переменными). Это означает, что единственный результат выполнения функции — это возвращаемое значение. А единственное, что может повлиять на возвращаемое значение — это аргументы, передаваемые в функцию.

Вот она, голубая мечта unit-тестеров. Можно протестировать каждую функцию в программе используя только нужные аргументы. Нет необходимости вызывать функции в правильном порядке или воссоздавать правильное внешнее состояние. Всё что вам нужно, это передать аргументы, которые соответствуют граничным случаям. Если все функции в вашей программе проходят Unit-тесты, то вы можете быть намного более уверены в качестве вашего ПО, чем в случае императивных языков программирования. В Java или C++ проверки возвращаемого значения не достаточно — функция может поменять внешнее состояние, которое тоже подлежит проверке. В ФП такой проблемы нет.

Отладка

Если функциональная программа ведёт себя не так, как вы ожидаете, то отладка — это пара пустяков. Вы всегда можете воспроизвести проблему, потому что ошибка в функции не зависит от постороннего кода, который выполнялся ранее. В императивной программе ошибка проявляется только на некоторое время. Вам придется пройти через ряд шагов, не относящихся к багу, из-за того, что работа функции зависит от внешнего состояния и побочных эффектов других функций. В ФП ситуация намного проще — если возвращаемое значение неправильное, то оно всегда будет неправильным, не зависимо от того, какие куски кода выполнялись прежде.

Как только вы воспроизведёте ошибку, найти её источник — тривиальная задача. Это даже приятно. Как только вы остановите выполнение программы, перед вами будет весь стек вызовов. Вы можете просмотреть аргументы вызова каждой функции, прямо как в императивном языке. С тем отличием, что в императивной программе этого не достаточно, ведь функции зависят от значений полей, глобальных переменных и состояний других классов. Функция в ФП зависит только от своих аргументов, и эта информация оказывается прямо у вас перед глазами! Даже больше, в императивной программе проверки возвращаемого значения не достаточно для того, чтобы сказать, правильно ли ведёт себя кусок кода. Вам придётся выследить десятки объектов за пределами функции, чтобы удостовериться, что всё работает правильно. В функциональном программировании всё, что нужно сделать — это взглянуть на возвращаемое значение!

Проходясь по стеку, вы обращаете внимание на передаваемые аргументы и возвращаемые значения. Как только возвращаемое значение отклоняется от нормы, вы углубляетесь в функцию и двигаетесь дальше. Так повторяется несколько раз пока вы не найдёте источник ошибки!

Многопоточность

Функциональная программа сразу готова к распараллеливанию без каких-либо изменений. Вам не придётся задумываться о deadlock-ах или состояниях гонки (race conditions) потому что вам не нужны блокировки! Ни один кусочек данных в функциональной программе не меняется дважды одним и тем же потоком или разными. Это означает, что вы можете легко добавить потоков к вашей программе даже не задумываясь при этом о проблемах, присущих императивным языкам.

Если дела обстоят подобным образом, то почему так редко функциональные языки программирования используются в многопоточных приложениях? На самом деле чаще, чем вы думаете. Компания Ericsson разработала функциональный язык под названием Erlang для использования на отказоустойчивых и масштабируемых телекоммуникационных коммутаторах. Многие отметили преимущества Erlang-а и стали его использовать. Мы говорим о телекоммуникациях и системах контроля трафика, которые далеко не так просто масштабируются, как типичные системы, разработанные на Wall Street. Вообще-то, системы написанные на Erlang, не такие масштабируемые и надёжные, как Java системы. Erlang системы просто сверхнадёжные.

На этом история многопоточности не заканчивается. Если вы пишете по сути однопоточное приложение, то компилятор всё равно может оптимизировать функциональную программу так, чтобы она использовала несколько CPU. Посмотрим на следующий кусок кода.

```
String s1 = somewhatLongOperation1();
String s2 = somewhatLongOperation2();
String s3 = concatenate(s1, s2);
```

Компилятор функционального языка может проанализировать код, классифицировать функции, которые создают строки `s1` и `s2`, как функции потребляющие много времени, и запустить их параллельно. Это невозможно сделать в императивном языке, потому что каждая функция может изменять внешнее состояние и код, идущий непосредственно после вызова, может зависеть от неё. В ФП автоматический анализ функций и поиск подходящих кандидатов для распараллеливания — это тривиальнейшая задача, как автоматический `inline`! В этом смысле функциональный стиль программирования соответствует требованиям завтрашнего дня. Разработчики железа уже не могут заставить CPU работать быстрее. Вместо этого они наращивают количество ядер и заявляют о четырёхкратном увеличении скорости многопоточных вычислений. Конечно они очень вовремя забывают сказать, что ваш новый процессор покажет прирост только в программах, разработанных с учётом распараллеливания. Среди императивного ПО таких очень мало. Зато 100% функциональных программ готовы к многопоточности из коробки.

Развёртывание по горячему

В старые времена для установки обновлений Windows приходилось перезагружать компьютер. Много раз. После установки новой версии медиа проигрывателя. В Windows XP произошли значительные изменения, но ситуация всё ещё далека от идеальной (сегодня я запустил Windows Update на работе и теперь надоедливое напоминание не оставит меня в покое, пока

не перезагружусь). В Unix системах модель обновления была получше. Для установки обновлений приходилось останавливать некоторые компоненты, но не всю ОС. Хотя ситуация выглядит лучше, но для большого класса серверных приложений это всё ещё не приемлемо. Телекоммуникационные системы должны быть включены 100% времени, ведь если из-за обновления человек не сможет вызвать скорую, то жизни могут быть потеряны. Фирмы с Wall Streets тоже не желают останавливать сервера на выходных, чтобы установить обновления.

В идеале нужно обновить все нужные участки кода не останавливая систему в принципе. В императивном мире это невозможно [пер. в Smalltalk-е очень даже возможно]. Представьте себе выгрузку Java класса на лету и перезагрузка новой версии. Если бы мы так сделали, то все экземпляры класса стали бы нерабочими, потому что потерялось бы состояние, которое они хранили. Нам пришлось бы писать хитрый код, для контроля версий. Пришлось бы сериализовать все созданные экземпляры класса, потом уничтожить их, создать экземпляры нового класса, попытаться загрузить сериализованные данные в надежде, что миграция пройдёт нормально и новые экземпляры будут валидными. И кроме того, миграционный код необходимо писать каждый раз вручную. И ещё миграционный код должен сохранять ссылки между объектами. В теории ещё куда ни шло, но на практике это никогда не заработает.

В функциональной программе всё состояние хранится в стеке в виде аргументов функций. Это позволяет значительно упростить развёртывание по горячему! По сути всё что нужно сделать — это вычислить разницу между кодом на рабочем сервере и новой версией, и установить изменения в коде. Остальное будет сделано языковыми инструментами автоматически! Если вы думаете, что это научная фантастика, то дважды подумайте. Инженеры, имеющие дело с Erlang, годами обновляют свои системы без остановки их работы.

Доказательные вычисления и оптимизация (Machine Assisted Proofs and Optimizations)

Еще одно интересное свойство функциональных языков программирования состоит в том, что их можно изучать с математической точки зрения. Так как функциональный язык — это реализация формальной системы, то все математические операции используемые на бумаге, могут быть применены и к функциональным программам. Компилятор, например, может конвертировать участок кода в эквивалентный, но более эффективный кусок, при этом математически обосновав их эквивалентность [7]. Реляционные базы данных годами производят такие оптимизации. Ничто не мешает использовать аналогичные приёмы в обычных программах.

Дополнительно вы можете использовать математический аппарат, чтобы доказать корректность участков ваших программ. При желании можно написать инструменты, которые анализируют код и автоматически создают Unit-тесты для граничных случаев! Такая функциональность бесценна для сверхнадёжных систем (rock solid systems). При разработке систем контроля кардиостимуляторов или управления воздушным трафиком такие инструменты просто необходимы. Если же ваши разработки не находятся в сфере критически важных приложений, то инструменты автоматической проверки всё равно дадут вам гигантское преимущество перед вашими конкурентами.

Функции высшего порядка

Помните, когда я говорил о преимуществах ФП, я отметил, что «всё выглядит красиво, но бесполезно, если мне придётся писать на корявом языке, в котором всё `final`». Это было заблуждением. Использование `final` повсеместно выглядит коряво только в императивных языках программирования, таких как Java. Функциональные языки программирования оперируют другими видами абстракций, такими, что вы забудете о том, что когда-то любили менять переменные. Один из таких инструментов — это функции высшего порядка.

В ФП функция — это не тоже самое, что функция в Java или C. Это надмножество — они могут тоже самое, что Java функции и даже больше. Пусть у нас есть функция на C:

```
int add(int i, int j) {
    return i + j;
}
```

В ФП это не тоже самое, что обычная C функция. Давайте расширим наш Java компилятор, чтобы он поддерживал такую запись. Компилятор должен превратить объявление функции в следующий Java код (не забывайте, что везде присутствует неявный `final`):

```
class add_function_t {
    int add(int i, int j) {
```

```
        return i + j;
    }
}

add_function_t add = new add_function_t();
```

Символ add не совсем функция. Это маленький класс с одним методом. Теперь мы можем передавать add в качестве аргумента в другие функции. Мы можем записать его в другой символ. Мы можем создавать экземпляры `add_function_t` в runtime и они будут уничтожены сборщиком мусора, если станут ненужными. Функции становятся базовыми объектами, как числа и строки. Функции, которые оперируют функциями (принимают их в качестве аргументов) называются функциями высшего порядка. Пусть это вас не пугает. Понятие функций высшего порядка почти не отличается от понятия Java классов, которые оперируют друг другом (мы можем передавать классы в другие классы). Мы можем называть их «классы высшего порядка», но никто этим не заморачивается, потому что за Java не стоит строгое академическое сообщество.

Как и когда нужно использовать функции высшего порядка? Я рад, что вы спросили. Вы пишите свою программу как один большой монолитный кусок кода не заботясь об иерархии классов. Если вы увидите, что какой-то участок кода повторяется в разных места, вы выносите его в отдельную функцию (к счастью в школах еще учат как это делать). Если вы замечаете, что часть логики в вашей функции должна вести себя по разному в некоторых ситуациях, то вы создаёте функцию высшего порядка. Запутались? Вот реальный пример из мой работы.

Предположим, что у нас есть участок Java кода, который получает сообщение, преобразует его различными способами и передаёт на другой сервер.

```
    void handleMessage(Message msg) {
        // ...
        msg.setClientCode("ABCD_123");
        // ...

        sendMessage(msg);
    }

    // ...
}
```

Теперь представьте себе, что система поменялась, и теперь нужно распределять сообщения между двумя серверами вместо одного. Всё остаётся неизменным, кроме кода клиента — второй сервер хочет получать этот код в другом формате. Как нам справиться с этой ситуацией? Мы можем проверять, куда должно попасть сообщение, и в зависимости от этого устанавливать правильный код клиента. Например так:

```
class MessageHandler {
    void handleMessage(Message msg) {
        // ...
        if(msg.getDestination().equals("server1") {
            msg.setClientCode("ABCD_123");
        } else {
            msg.setClientCode("123_ABC");
        }
        // ...

        sendMessage(msg);
    }

    // ...
}
```

Но такой подход плохо масштабируется. При добавлении новых серверов функция будет расти линейно, и внесение изменений превратится в кошмар. Объектно ориентированный подход заключается в выделении общего суперкласса `MessageHandler` и вынесение логики определения кода клиента в подклассы:

```
abstract class MessageHandler {
    void handleMessage(Message msg) {
```

```
        // ...
        msg.setClientCode(getClientCode());
        // ...

        sendMessage(msg);
    }

    abstract String getClientCode();

    // ...
}


class MessageHandlerOne extends MessageHandler {
    String getClientCode() {
        return "ABCD_123";
    }
}


class MessageHandlerTwo extends MessageHandler {
    String getClientCode() {
        return "123_ABCD";
    }
}
```

Теперь для каждого сервера мы можем создать экземпляр соответствующего класса. Добавление новых сервером становится более удобным. Но для такого небольшого изменения многовато текста. Пришлось создать два новых типа чтобы просто добавить поддержку различного кода клиента! Теперь сделаем тоже самое в нашем языке с поддержкой функций высшего порядка:

```
class MessageHandler {
    void handleMessage(Message msg, Function getClientCode) {
        // ...
        Message msg1 = msg.setClientCode(getClientCode());
        // ...

        sendMessage(msg1);
    }

    // ...
}


String getClientCodeOne() {
    return "ABCD_123";
}


String getClientCodeTwo() {
    return "123_ABCD";
}


MessageHandler handler = new MessageHandler();
handler.handleMessage(someMsg, getClientCodeOne);
```

Мы не создавали новых типов и не усложняли иерархию классов. Мы просто передали функцию в качестве параметра. Мы достигли того же эффекта, как и в объектно-ориентированном аналоге, только с некоторыми преимуществами. Мы не привязывали себя к какой-либо иерархии классов: мы можем передавать любые другие функции в runtime и менять их в любой момент, сохраняя при этом высокий уровень модульности меньшим количеством кода. По сути компилятор создал объектно-ориентированный «клей» вместо нас! При этом сохраняются все остальные преимущества ФП. Конечно абстракции, предлагаемые функциональными языками на этом не заканчиваются. Функции высшего порядка это только начало

Каррирование

Большинство людей, с которыми я встречаюсь, прочли книгу «Паттерны проектирования» Банды Четырёх. Любой уважающий себя программист будет говорить, что книга не привязана к какому-либо конкретному языку программирования, а паттерны применимы к разработке ПО в целом. Это благородное заявление. Но к сожалению оно далеко от истины.

Функциональные языки невероятно выразительны. В функциональном языке вам не понадобятся паттерны проектирования, потому что язык настолько высокоуровневый, что вы легко начнёте программировать в концепциях, которые исключают все известные паттерны программирования. Одним из таких паттернов является Адаптер (чем он отличается от Фасада? Похоже, что кому-то понадобилось наштамповать побольше страниц, чтобы выполнить условия контракта). Этот паттерн оказывается ненужным если в языке есть поддержка каррирования.

Паттерн Адаптер наиболее часто применяется к «стандартной» единице абстракции в Java — классу. В функциональных языках паттерн применяется к функциям. Паттерн берёт интерфейс и преобразует его в другой интерфейс, согласно определённым требованиям. Вот пример паттерна Адаптер:

```
int pow(int i, int j);
int square(int i)
{
    return pow(i, 2);
}
```

Этот код адаптирует интерфейс функции, возводящей число в произвольную степень, к интерфейсу функции, которая возводит число в квадрат. В аккадемических кругах этот простейший приём называется каррирование (в честь специалиста по логике Хаскелла Карри (Haskell Curry), который провёл ряд математических трюков, чтобы всё это формализовать). Так как в ФП функции используются повсеместно в качестве аргументов, каррирование используется очень часто, чтобы привести функции к интерфейсу, необходимому в том или ином месте. Так как интерфейс функции — это её аргументы, то каррирование используется для уменьшения количества аргументов (как в примере выше).

Этот инструмент является встроенным в функциональные языки. Вам не нужно вручную создавать функцию, которая оборачивает оригинал. Функциональный язык сделает всё за вас. Как обычно давайте расширим наш язык, добавив в него каррирование.

```
square = int pow(int i, 2);
```

Этой строкой мы автоматически создаём функцию возведения в квадрат с одним аргументом. Новая функция будет вызывать функцию `pow`, подставляя 2 в качестве второго аргумента. С точки зрения Java, это будет выглядеть следующим образом:

```
class square_function_t {
    int square(int i) {
        return pow(i, 2);
    }
}
square_function_t square = new square_function_t();
```

Как видите, мы просто написали обёртку над оригинальной функцией. В ФП каррирование как раз и представляет из себя простой и удобный способ создания обёрток. Вы сосредотачиваетесь на задаче, а компилятор пишет необходимый код за вас! Всё очень просто, и происходит каждый раз, когда вы хотите использовать паттерн Адаптер (обёртку).

Ленивые вычисления

Ленивые (или отложенные) вычисления — это интересная техника, которая становится возможной как только вы усвоите функциональную философию. Мы уже встречали следующий кусок кода, когда говорили о многопоточности:

```
String s1 = somewhatLongOperation1();
String s2 = somewhatLongOperation2();
String s3 = concatenate(s1, s2);
```

В императивных языках программирования очерёдность вычисления не вызывает никаких вопросов. Поскольку каждая функция может повлиять или зависеть от внешнего состояния, то необходимо соблюдать чёткую очерёдность вызовов: сначала `somewhatLongOperation1`, затем `somewhatLongOperation2`, и `concatenate` в конце. Но не всё так просто в функциональных языках.

Как мы уже видели ранее `somewhatLongOperation1` и `somewhatLongOperation2` могут быть запущены одновременно, потому что функции гарантированно не влияют и не зависят от глобального состояния. Но что, если мы не хотим выполнять их одновременно, нужно ли вызывать их последовательно? Ответ — нет. Эти вычисления должны быть запущены, только если какая-либо другая функция зависит от `s1` и `s2`. Нам даже не нужно выполнять их до тех пор, пока они понадобятся внутри `concatenate`. Если вместо `concatenate` мы подставим функцию, которая в зависимости от условия использует один аргумент из двух, то второй аргумент можно даже не вычислять! Haskell — это пример языка с отложенными вычислениями. В Haskell отсутствует гарантия какой-либо очередности вызовов (вообще!), потому что Haskell выполняет код по мере необходимости.

Ленивые вычисления обладают рядом достоинств как и некоторыми недостатками. В следующем разделе мы обсудим достоинства и я объясню как уживаться с недостатками.

Оптимизация

Ленивые вычисления обеспечивают громадный потенциал для оптимизаций. Ленивый компилятор рассматривает код в точности как математик изучает алгебраические выражения — он может отменять некоторые вещи, отменять выполнение тех или иных участков кода, менять очерёдность вызовов для большей эффективности, даже располагать код таким образом, чтобы уменьшить количество ошибок, при этом гарантируя целостность программы. Это самое большое преимущество при описании программы строгими формальными примитивами — код подчиняется математическим законам и может быть изучен математическими методами.

Абстрагирование структур управления

Ленивые вычисления обеспечивают настолько высокий уровень абстракций, что становятся возможными удивительные вещи. Например, представим себе реализацию следующей управляющей структуры:

```
unless(stock.isEuropean()) {
    sendToSEC(stock);
}
```

Мы хотим, чтобы функция `sendToSEC` выполнялась только если фонд (stock) не европейский. Как можно реализовать `unless`? Без ленивый вычислений нам бы понадобилась система макросов, но в языках, подобных Haskell, это не обязательно. Мы можем объявить `unless` в виде функции!

```
void unless(boolean condition, List code) {
    if(!condition)
        code;
}
```

Заметьте, что `code` не будет выполняться, если `condition == true`. В строгих языках такое поведение невозможно повторить, так как аргументы будут вычислены прежде, чем `unless` будет вызвана.

Бесконечные структуры данных

Ленивые языки позволяют создавать бесконечные структуры данных, создание которых в строгих языках гораздо сложнее [пер. — только не в Python]. Например представьте себе последовательность Фибоначи. Очевидно, что мы не можем вычислить бесконечный список за конечное время и при этом сохранить его в памяти. В строгих языках, таких как Java, мы просто написали бы функцию, которая возвращает произвольный член последовательности. В языках подобных Haskell мы

можем абстрагироваться и просто объявить бесконечный список чисел Фибоначи. Так как язык ленивый, то будут вычислены лишь необходимые части списка, которые реально используются в программе. Это позволяет абстрагироваться от большого числа проблем и посмотреть на них с более высокого уровня (например можно использовать функции обработки списков на бесконечных последовательностях).

Недостатки

Конечно бесплатный сыр бывает только в мышеловке. Ленивые вычисления тянут за собой ряд недостатков. В основном это недостатки от лени. В реальности очень часто нужен прямой порядок вычислений. Возьмём, например, следующий код:

```
System.out.println("Please enter your name: ");
System.in.readLine();
```

В ленивом языке никто не гарантирует, что первая строка выполнится раньше второй! Это означает, что мы не можем делать ввод-вывод, не можем нормально использовать нативные функции (ведь их нужно вызывать в определённом порядке, чтобы учитывать их побочные эффекты), и не можем взаимодействовать с внешним миром! Если мы введём механизм для упорядочивания выполнения кода, то потеряем преимущество математической строгости кода (а следом потеряем все плюшки функционального программирования). К счастью ещё не всё потеряно. Математики взялись за работу и придумали несколько приёмов для того, чтобы убедится в правильном порядке выполняемых инструкций не потеряв функционального духа. Мы получили лучшее от двух миров! Такие приёмы включают в себя продолжения (continuation), монады (monads) и однозначная типизация (uniqueness typing). В данной статье мы поработаем с продолжениями, а монады и однозначную типизацию отложим до следующего раза. Занятно, что продолжения очень полезная штука, которая используется не только для задания строгого порядка вычислений. Об этом мы тоже поговорим.

Продолжения

Продолжения в программировании играют такую же роль, как «Код да Винчи» в человеческой истории: удивительное разоблачение величайшей тайны человечества. Ну, может не совсем так, но они точно срывают покровы, как в своё время вы научились брать корень из -1.

Когда мы рассматривали функции, мы изучили лишь половину правды, ведь мы исходили из предположения, что функция возвращает значение в вызывающую её функцию. В этом смысле продолжение — это обобщение функций. Функция не обязательно должна возвращать управление в то место, откуда её вызвали, а может возвращать в любое место программы. «Продолжение» — это параметр, который мы можем передать в функцию, чтобы указать точку возврата. Звучит намного страшнее, чем есть на самом деле. Давайте взглянем на следующий код:

```
int i = add(5, 10);
int j = square(i);
```

Функция `add` возвращает число 15, которое записывается в `i`, в том месте, где функция и была вызвана. Затем значение `i` используется при вызове `square`. Заметьте, что ленивый компилятор не может поменять очередность вычислений, ведь вторая строка зависит от результата первой. Мы можем переписать этот код с использованием Стиль Передачи Продолжения (Continuation Passing Style или CPS), когда `add` возвращает значение в функцию `square`.

```
int j = add(5, 10, square);
```

В таком случае `add` получает дополнительный аргумент — функцию, которая будет вызвана после того, как `add` закончит работать. В обоих примерах `j` будет равен 225.

В этом и заключается первый приём, позволяющий задать порядок выполнения двух выражений. Вернёмся к нашему примеру с вводом-выводом

```
System.out.println("Please enter your name: ");
System.in.readLine();
```

Эти две строки не зависят друг от друга, и компилятор волен поменять их порядок по своему хотению. Но если мы перепишем в CPS, то тем самым добавим нужную зависимость, и компилятору придётся проводить вычисления одно за другим!

```
System.out.println("Please enter your name: ", System.in.readLine);
```

В таком случае `println` должен будет вызвать `readLine`, передав ему свой результат, и вернуть результат `readLine` в конце. В таком виде мы можем быть уверены, что эти функции будут вызваны по очереди, и что `readLine` вообще вызовется (ведь компилятор ожидает получить результат последней операции). В случае Java `println` возвращает `void`. Но если бы возвращалось какое-либо абстрактное значение (которое может служить аргументом `readLine`), то это решило бы нашу проблему! Конечно выстраивание таких цепочек функций сильно ухудшает читаемость кода, но с этим можно бороться. Мы можем добавить в наш язык синтаксических плюшек, которые позволят нам писать выражения как обычно, а компилятор автоматически выстраивал бы вычисления в цепочки. Теперь мы можем проводить вычисления в любом порядке, не потеряв при этом достоинств ФП (включая возможность исследовать программу математическими методами)! Если вас это сбивает с толку, то помните, что функции — это всего лишь экземпляры класса с единственным членом. Перепишите наш пример так, чтобы `println` и `readLine` были экземплярами классов, так вам станет понятней.

Но на этом польза продолжений не заканчивается. Мы можем написать всю программу целиком используя CPS, чтобы каждая функция вызывалась с дополнительным параметром, продолжением, в которое передаётся результат. В принципе любую программу можно перевести на CPS, если воспринимать каждую функцию как частный случай продолжений. Такое преобразование можно произвести автоматически (в действительности многие компиляторы так и делают).

Как только мы переведём программу к CPS виду, становится ясно, что у каждой инструкции есть продолжение, функция в которую будет передаваться результат, что в обычной программе было бы точкой вызова. Возьмём любую инструкцию из последнего примера, например `add(5,10)`. В программе, написанной в CPS виде, понятно что будет являться продолжением — это функция, которую `add` вызовет по окончанию работы. Но что будет продолжением в случае не-CPS программы? Мы, конечно, можем конвертировать программу в CPS, но нужно ли это?

Оказывается, что в этом нет необходимости. Посмотрите внимательно на наше CPS преобразование. Если вы начнёте писать компилятор для него, то обнаружите, что для CPS версии не нужен стек! Функции никогда ничего не возвращают, в традиционном понимании слова «return», они просто вызывают другую функцию, подставляя результат вычислений. Отпадает необходимость проталкивать (push) аргументы в стек перед каждым вызовом, а потом извлекать (pop) их обратно. Мы можем просто хранить аргументы в каком-либо фиксированном участке памяти и использовать `jump` вместо обычного вызова. Нам нет нужны хранить первоначальные аргументы, ведь они больше никогда не понадобятся, ведь функции ничего не возвращают!

Таким образом, программы в CPS стиле не нуждаются в стеке, но содержат дополнительный аргумент, в виде функции, которую нужно вызвать. Программы в не-CPS стиле лишены дополнительного аргумента, но используют стек. Что же хранится в стеке? Просто аргументы и указатель на участок памяти, куда должна вернуться функция. Ну как, вы уже догадались? В стеке храниться информация о продолжениях! Указатель на точку возврата в стеке — это то же самое, что и функция, которую нужно вызвать, в CPS программах! Чтобы выяснить, какое продолжение у `add(5,10)`, достаточно взять из стека точку возврата.

Это было не трудно. Продолжение и указатель на точку возврата — это действительно одно и то же, только продолжение указывается явно, и по этому оно может отличаться от того места, где функция была вызвана. Если вы помните, что продолжение — это функция, а функция в нашем языке компилируется в экземпляр класса, то поймёте, что указатель на точку возврата в стеке и указатель на продолжение — это в действительности одно и то же, ведь наша функция (как экземпляр класса) — это всего лишь указатель. А значит, что в любой момент времени в вашей программы вы можете запросить текущее продолжение (по сути информацию из стека).

Хорошо, теперь мы уяснили, что же такое текущее продолжение. Что это значит? Если мы возьмём текущее продолжение и сохраним его где-нибудь, мы тем самым сохраним текущее состояние программы — заморозим её. Это похоже на режим гибернации ОС. В объекте продолжения хранится информация, необходимая для возобновления выполнения программы с той точки, когда был запрошен объект продолжения. Операционная система постоянно так делает с вашими программами, когда переключает контекст между потоками. Разница лишь в том, что всё находится под контролем ОС. Если вы запросите объект продолжения (в Scheme это делается вызовом функции `call-with-current-continuation`), то вы получите объект с текущим продолжением — стеком (или в случае CPS — функцией следующего вызова). Вы можете сохранить этот объект в переменную (или даже на диск). Если вы решите «перезапустить» программу с этим продолжением, то состояние вашей программы «преобразуется» к состоянию на момент взятия объекта продолжения. Это то же самое, как переключение к

приостановленному потоку, или пробуждение ОС после гибернации. С тем исключением, что вы можете проделывать это много раз подряд. После пробуждения ОС информация о гибернации уничтожается. Если этого не делать, то можно было бы восстанавливать состояние ОС с одной и той же точки. Это почти как путешествие по времени. С продолжениями вы можете себе такое позволить!

В каких ситуациях продолжения будут полезны? Обычно если вы пытаетесь эмулировать состояние в системах лишенных такового по сути. Отличное применение продолжения нашли в Web-приложениях (например во фреймворке Seaside для языка Smalltalk). ASP.NET от Microsoft прикладывает огромные усилия, чтобы сохранять состояние между запросами, и облегчить вам жизнь. Если бы C# поддерживал продолжения, то сложность ASP.NET можно было бы уменьшить в два раза — достаточно было бы сохранять продолжение и восстанавливать его при следующем запросе. С точки зрения Web-программиста не было бы ни единого разрыва — программа продолжала бы свою работу со следующей строки! Продолжения — невероятно полезная абстракция для решения некоторых проблем. Учитывая то, что всё больше и больше традиционных толстых клиентов перемещаются в Web, важность продолжений будет со временем только расти.

Сопоставление с образцом (Pattern matching)

Сопоставление с образцом не такая уж новая или инновационная идея. На самом деле она имеет слабое отношение к функциональному программированию. Единственная причина, по которой его часто связывают с ФП, это то, что с некоторых пор в функциональных языках есть сопоставление с образцом, а в императивных — нет.

Давайте начнём наше знакомство с Pattern matching следующим примером. Вот функция вычисления чисел Фибоначи на Java:

```
int fib(int n) {
    if(n == 0) return 1;
    if(n == 1) return 1;

    return fib(n - 2) + fib(n - 1);
}
```

А вот пример на Java-подобном языке с поддержкой Pattern matching-а

```
int fib(0) {
    return 1;
}
int fib(1) {
    return 1;
}
int fib(int n) {
    return fib(n - 2) + fib(n - 1);
}
```

В чём разница? Компилятор реализует ветвление за нас.

Подумаешь, велика важность! Действительно важность не велика. Было подмечено, что большое количество функций содержат сложные `switch` конструкции (это отчасти верно для функциональных программ), и было принято решение выделить этот момент. Определение функции разбивается на несколько вариантов, и устанавливается паттерн на месте аргументов функции (это напоминает перегрузку методов). Когда происходит вызов функции, компилятор на лету сравнивает аргументы со всеми определениями и выбирает наиболее подходящий. Обычно выбор падает на самое специализированное определение функции. Например `int fib(int n)` может быть вызвана при `n` равном 1, но не будет, ведь `int fib(1)` — более специализированное определение.

Сопоставление с образцом обычно выглядит сложнее, чем в нашем примере. Например сложная система Pattern matching позволяет писать следующий код:

```
int f(int n < 10) { ... }
int f(int n) { ... }
```

Когда сопоставление с образцом может быть полезно? Список таких случаев на удивление очень большой! Каждый раз,

когда вы используете сложные конструкции вложенных `if`, pattern matching может справиться лучше с меньшим количеством кода. В голову приходит хороший пример с функцией `WndProc`, которая реализуется в каждой Win32 программе (даже если она спрятана от программиста за высоким забором абстракций). Обычно сопоставление с образцом может даже проверять содержимое коллекций. Например, если вы передаёте массив в функцию, то вы можете отбирать все массивы, у которых первый элемент равен 1, а третий элемент больше 3.

Ещё одним преимуществом Pattern matching является то, что в случае внесения изменений вам не придётся копаться в одной огромной функции. Вам достаточно будет добавить (или изменить) некоторые определения функций. Тем самым мы избавляется от целого пласта паттернов из знаменитой книги Банды Четырёх. Чем сложнее и ветвистее условия, тем полезнее будет использовать Pattern matching. Как только вы начнёте их использовать, то удивитесь, как вы могли раньше без них обходится.

Замыкания

До сих пор мы обсуждали особенности ФП в контексте «чисто» функциональных языков — языков, которые являются реализацией лямбда исчисления и не содержат особенностей, противоречащих формальной системе Чёрча. Тем не менее, многие черты функциональных языков используются за пределами лямбда исчисления. Хотя реализация аксиоматической системы интересна с точки зрения программирования в терминах математических выражений, это не всегда может быть применимо на практике. Многие языки предпочитают использовать элементы функциональных языков не придерживаясь строгой функциональной доктрины. Некоторые такие языки (например Common Lisp) не требуют от переменных быть `final` — их значения можно менять. Они даже не требуют, чтобы функции зависели только от своих аргументов — функциям дозволенно обращаться к состоянию за пределом своей области видимости. Но при этом они включают в себя такие особенности, как функции высшего порядка. Передача функции в не-чистом языке немного отличается от аналогичной операции в пределах лямбда исчисления и требует наличия интересной особенности под названием: лексическое замыкание. Давайте взглянем на следующий пример. Помните, что в данном случае переменные не `final` и функция может обращаться к переменным за пределом своей области видимости:

```
Function makePowerFn(int power) {
   int powerFn(int base) {
       return pow(base, power);
   }


   return powerFn;
}


Function square = makePowerFn(2);
square(3); // returns 9
```

Функция `make-power-fn` возвращает функцию, которая принимает один аргумент и возводит его в определённую степень. Что произойдёт, когда мы попробуем вычислить `square(3)`? Переменная `power` находится вне области видимости `powerFn`, потому что `makePowerFn` уже завершилась, и её стек уничтожен. Как же тогда работает `square`? Язык должен каким-либо образом сохранить значение `power`, чтобы функция `square` могла работать. А что если мы создадим ещё одну функцию `cube`, которая возводит число в третью степень? Язык должен будет сохранять два значения `power` для каждой созданной в `make-power-fn` функции. Феномен хранения этих значений и называется замыканием. Замыкание не только сохраняет аргументы верхней функции. Например замыкание может выглядеть следующим образом:

```
Function makeIncrementer() {
   int n = 0;


   int increment() {
       return ++n;
   }
}


Function inc1 = makeIncrementer();
Function inc2 = makeIncrementer();
```

```
inc1(); // returns 1;
inc1(); // returns 2;
inc1(); // returns 3;
inc2(); // returns 1;
inc2(); // returns 2;
inc2(); // returns 3;
```

В процессе выполнения значения `n` сохраняются, и счётчики имеют доступ к ним. Более того у каждого счётчика своя копия `n`, не смотря на то, что они должны были исчезнуть после того, как функция `makeIncrementer` отработает. Как же компилятор умудряется это скомпилировать? Что происходит за кулисами замыканий? К счастью у нас есть волшебный пропуск.

Всё сделано достаточно логично. С первого взгляда ясно, что локальные переменные больше не подчиняются правилам области видимости и их время жизни не определено. Очевидно, что они больше не хранятся в стеке — их нужно держать в куче (heap) [8]. Замыкание, следовательно, сделано как обычная функция, которую мы обсуждали ранее, за исключением того, что в нём есть дополнительная ссылка на окружающие переменные:

```
class some_function_t {
   SymbolTable parentScope;

   // ...
}
```

Если замыкание обращается к переменной, которой нет в локальной области видимости, тогда оно принимает во внимание родительскую область. Вот и всё! Замыкание связывает функциональный мир с миром ООП. Каждый раз, когда вы создаёте класс, который хранит некоторое состояние, и передаёте его куда-то, вспомните про замыкания. Замыкание — это всего лишь объект, который создаёт «атрибуты» на лету, забирая их из области видимости, чтобы вам не пришлось делать это самим.

Что теперь?

Эта статья проходится лишь по верхушке айсберга Функционального Программирования. Вы можете копнуть глубже и увидеть нечто действительно большое, а в нашем случае ещё и хорошее. В будущем я планирую написать о теории категорий, монадах, функциональных структурах данных, системе типов в функциональных языках, функциональной многопоточности, функциональных базах данных, и ещё о многих вещах. Если у меня получится написать (и изучить в процессе) хотя бы о половине из этих тем, моя жизнь пройдёт не зря. А пока, Google — ваш верный друг.

Комментарии?

Если у вас есть вопросы, комментарии или предложения, черканите записочку на адрес coffeemug [собачка] gmail.com. Буду рад любым вашим отзывам.

Примечания

[1]. Когда я искал работу осенью 2005 года, я частенько задавал этот вопрос. Было очень занятно видеть пустой взгляд вместо ответа. Я-то думал, что за зарплату $300,000 у этих людей должно быть хорошее понимание всех доступных инструментов.

[2]. Этот вопрос неоднозначный. Физики и математики вынуждены признать, что нет ничего предельно ясного во вселенной, что можно было бы описать математически.

[3]. Я ненавидел уроки истории, которые предлагали только сухую хронологию дат, имён и событий. Для меня история — это жизни людей, которые изменили мир. Это те их личные причины, которая стоят за их действиями, и механизмы, которыми они оказывали влияние на миллионы душ. По этой причине исторический раздел этой статьи безнадёжно неполон. Здесь описываются только наиболее значимые люди и события.

[4]. Когда я только начинал изучать функциональное программирование, меня очень нервировал термин «лямбда», потому что я не мог до конца понять, что же он обозначает. В данном контексте лямбда — это функция. А греческая буква используется для удобства математической записи. Каждый раз, когда вы слышите «лямбда» в разговоре о функциональном программировании, переводите это про себя в «функцию».

[5]. Занимательно, что строки в Java не изменяются. Интересно было бы выяснить причину такого вероломства, но не будем отвлекаться.

[6]. Большинство компиляторов функциональных языков умеют оптимизировать рекурсивные функции превращая их в циклы по мере возможностей. Это называется оптимизация хвостовой рекурсии.

[7]. Обратное не всегда верно. Если иногда возможно доказать эквивалентность двух участков кода, но в общем случае это не возможно.

[8]. На самом деле это не медленнее, чем хранить в стеке, поскольку при использовании сборщика мусора выделение памяти занимает O(1) операций.

Теги: функциональное программирование, ликбез, история

Хабы: Программирование

+174 2130 307k 151 Поделиться

115,5 Карма 0,0 Рейтинг

Щекн-Итрч @sheknitrtch
Пользователь

ПОХОЖИЕ ПУБЛИКАЦИИ

15 февраля 2021 в 15:19
Сильные стороны функционального программирования
20 8k 62 48

25 декабря 2020 в 15:15
Функциональное программирование на TypeScript: паттерн «класс типов»
19 4,9k 72 25

1 ноября 2020 в 19:56
Функциональное программирование на TypeScript: полиморфизм родов высших порядков
61 16,9k 166 59

ВАКАНСИИ

Middle | High middle front-end разработчик (React + Typescript)
от 80 000 до 160 000 ₽ · CSSSR · Можно удаленно

Backend Developer (Python/GO)
от 180 000 ₽ · AgentApp · Можно удаленно

C++ разработчик
от 1 500 $ · Toqomo · Санкт-Петербург · Можно удаленно

Reverse Engineer
от 3 500 до 4 000 $ · Hand2Note · Можно удаленно

C Разработчик (Embedded)
от 100 000 ₽ · Flipper Devices Inc. · Москва · Можно удаленно

Больше вакансий на Хабр Карьере

Реклама

Ad removed. Details

Комментарии 151

adaptun 19 апреля 2012 в 12:05 +17

Добавлю в папочку «Прочитать»

mishadoff 19 апреля 2012 в 13:36 +15

Ленивые вычисления? ;)

adaptun 19 апреля 2012 в 14:12 +13

Ох, ленивые.
The real benefit of lazy evaluation still has to be evaluated out.

Dimchansky 19 апреля 2012 в 15:46 +32

Прямо как на лекции по функциональному программированию:
— Сегодня я вам расскажу о ленивых вычислениях. Есть вопросы?

asci 19 апреля 2012 в 14:14 −5

А я осилил до середины, там про историю интересно, дальше в «Почитать»

adaptun 19 апреля 2012 в 14:18 +11

Бессонница. ФП. Тугие паруса.
Статью на хабре я прочёл до середины.

pinkey 20 апреля 2012 в 12:40 +3

Рабочий день. ФП. В оконце свет.
Вторую часть прочту в обед.

toxicdream 19 апреля 2012 в 14:18 +12

А я прочел...
Многое встало на свои места.
Автору и переводчику +100500.

HeartOfGold 19 апреля 2012 в 12:20 +2

Занимательно, что строки в Java не изменяются. Интересно было бы выяснить причину такого вероломства, но не будем отвлекаться.

Для перформанса памяти стринговые переменны лежат в пуле стрингов поэтому все переменные класса String с одинаковым значением могут ссылатья на один объект в памяти (при том что единственная другая связь между ними это единый JVM). Так как

для програмиста это неочевидно и отследить это проблематично то объект должен быть неизменным чтоб не возникло казусов что кто то привязал к твоему «Preved» своегое «Medved». Поэтому же String final чтоб от него не отнаследовали

catlion 19 апреля 2012 в 14:28 +5

В дотнете строки тоже immutable.

philpirj 21 апреля 2012 в 05:47 −1

Увы, от злоумышленного reflection'a не спасает.

flamingo 19 апреля 2012 в 12:27 +4

По-моему, это уже было на RSDN.

sheknitrtch 19 апреля 2012 в 12:30 +2

`site:rsdn.ru www.defmacro.org/ramblings/fp.html`

sheknitrtch 19 апреля 2012 в 12:31 +1

Сори, отправил раньше времени.
Поиском в Google `site:rsdn.ru "defmacro.org"` нашёл ссылки на другие статьи автора, но на эту — нет.

flamingo 19 апреля 2012 в 12:35 0

www.rsdn.ru/article/funcprog/fp.xml
Начало у этой статьи запоминающееся.

sheknitrtch 19 апреля 2012 в 12:37 +2

Таки да. Вы правы.
И что теперь делать с моим постом? Удалять?

soomrack 19 апреля 2012 в 12:40 +3

ИМХО. Поставь ссылку. Удалять не надо: ибо вдруг будет интересная дискуссия?

TheShock 19 апреля 2012 в 12:45 +2

Зачем ссылку? Есть ссылка на автора — зачем ссылка на другие переводы?

sheknitrtch 19 апреля 2012 в 12:46 +4

В переводе на RSDN есть довольно занимательные комментарии переводчика, который в ФП разбирается получше меня.

TheShock 19 апреля 2012 в 12:50 0

Ну с такой точки зрения — да, интересно.

flamingo 19 апреля 2012 в 12:41 0

Зачем? Может быть кто-то не читает RSDN, я констатировал факт. Пожалуй, перечитаю эту статью.

TheShock 19 апреля 2012 в 12:44 +10

Ни в коем случае. Вы перевели, а не скопипастили перевод, потому пускай будет

jinxal 20 апреля 2012 в 04:51 +1

Не надо удалять. Я бы и не узнал про статью если бы не ваш перевод.) Отличная статья, и перевод тоже очень качественный, прочитал с удовольствием.

sheknitrtch 19 апреля 2012 в 12:44 +3

Добавил в шапку ссылку на RSDN вариант перевода

soomrack 19 апреля 2012 в 12:35 +4

Классный текст! Правда я так и не понял когда имеет смысл использовать ФП, а когда — нет. В статье очень много слов посвящено тому, как на ФП можно программировать в смысле ООП. А все преимущества описаны как возможный потенциал: возможность распараллеливание, возможность оптимизации без участия человека.

Хотелось бы конкретных примеров:
1. Распараллеливание — берем известный алгоритм, пишем программу на ФП и на ООП так, как нам удобно и сравниваем.
2. Оптимизация — берем известный алгоритм, пишем программу на ФП и на ООП так, как нам удобно и сравниваем.
Я так понимаю, что на ФП код должен получиться существенно короче и понятней, скорость примерно одинаковой, так?

Относительно ФП у меня сложилось впечатление, что его преимущества проявляются при решении для больших задач. Для решения простых кусочков большой проблемы код пишется на ООП, а вот чтобы составить из этих кусочков решение всей проблемы стОит применять ФП.

НЛО прилетело и опубликовало эту надпись здесь

soomrack 19 апреля 2012 в 13:21 +1

А как в этом случае с переполнением стека и нагрузкой на каналы передачи данных? Не становится ли она сильно избыточной?

НЛО прилетело и опубликовало эту надпись здесь

soomrack 19 апреля 2012 в 13:32 +1

Когда у нас переменные не изменяются, то фактически это означает, что память одноразовая. И если бы работаем со сложной структурой, о которой думаем как о чем-то неделимом, то мы не можем изменить какой-то ее кусочек, мы обязаны менять ее целиком. Т.е. заводить новую и туда копировать данные. Это не создает чрезмерную нагрузку на память?

mishadoff 19 апреля 2012 в 13:35 0

В случае использования списков, такой проблемы не возникает

НЛО прилетело и опубликовало эту надпись здесь

sheknitrtch 19 апреля 2012 в 13:47 0

> А копирование памяти — операция довольно быстрая.

тем более если используется механизм copy-on-write

soomrack 19 апреля 2012 в 14:10 0

> Если мы работаем со сложной структурой, о которой думаем как о чем-то неделимом — то она должна быть неделимой вне зависимости от подхода.

Я неточно выразился. Возможно лучше s/неделимой/цельной/.

> А копирование памяти — операция довольно быстрая.

Все относительно. Но избыточного копирования больших объемов данных стараются избегать.

Впрочем, это не важно. Кажется я понял как сформулировать то, что меня напрягает в ФП: в ФП нет привычных структур данных, привычных в том смысле, что мы можем их схематично нарисовать на бумаге, и их хранение в памяти компьютера будет примерно таким же, как наша рукописная схема.

Пример: как в ФП будет хранится матрица целых чисел? Будет ли это непрерывный массив ячеек памяти длинных n^2? Если да, то на сколько эффективна будет работа с ним, когда нам потребуется изменять только несколько ячеек? Если нет, то насколько эффективна будет работа с ним, когда нам потребуется работать со всеми ячейками?

Возможно это покажется некорректным, когда в для реализации одних алгоритмов мы хотим понимать структуру данных как нечто целое и неделимое, а для других алгоритмов как — нечто составное. Но, наверное, в этом и состоит основная задача разработки удобных структур данных — чтобы они были удобные и допускали разные уровни абстракции.

nikitad 19 апреля 2012 в 15:16 0

Матрицу лучше реализовать на основе одноуровневого списка. Никаких проблем с выборкой и изменением не будет: можно написать что-то типа:

```
Matrix.select ( sub {
... // выбираем нужные индексы матрицы
$cc->( $result );
},
sub {
... // изменяем полученные индексы как нам надо
return $newMatrix;
} );
```

soomrack 19 апреля 2012 в 15:37 0

Можно. Но скорость работы с ней при будет удручающей.

nikitad 20 апреля 2012 в 15:56 0

Функциональные языки не предназначены для числодробилок — для этого есть фортран. Они лучше подходят для сложных, требующих высокого уровня абстракции задач, где скорость разработки важнее скорости работы.

MagicWolf 19 апреля 2012 в 18:14 0

По моему опыту с Clojure для этого случая лучше писать код с сайд-эффектами (в императивном стиле), но изолировать всю работу с состоянием внутри функции, которая будет чистой. Т.е. зависеть только от входа и не менять глобального состояния.

В итоге вы получите скорость mutable структур там, где это очень надо, и удобство функциональной разработки в остальных местах.

megalol 20 апреля 2012 в 00:12 +4

> Кажется я понял как сформулировать то, что меня напрягает в ФП: в ФП нет привычных структур данных, привычных в том смысле, что мы можем их схематично нарисовать на бумаге, и их хранение в памяти компьютера будет примерно таким же, как наша рукописная схема.

1. На бумаге датаориентированный подход виде трубы, которая видоизменяет данные намного проще нарисовать, как и понять, чем императивную блок-схему. На самом деле большая часть документации намного более функциональна, чем код, который она описывает.

2. Когдая вижу x_new = x_old + 1 я знаю, что в любом месте я могу сделать мысленную подстановку. Когда я вижу x = x + 1, мне нужно мысленного разделить код на две части — до присваивания и после. Это настолько очевидно, что сейчас в любом стилевом гайде любого языка говорится о том, что переиспользовать переменные нельзя. В этом случае в императивном языке присваиваний становится намного меньше, чем кажется.

> Пример: как в ФП будет хранится матрица целых чисел? Будет ли это непрерывный массив ячеек памяти длинных n^2? Если да, то на сколько эффективна будет работа с ним, когда нам потребуется изменять только несколько ячеек?

Матрица в ФП должна храниться как тип данных, у которого операции типа *, /, +, -, слайсов вызывают соответствующие функции библиотеки линейной алгебры, написанной на C.
На этом уровне абстракции требоваться изменять несколько ячеек придется очень редко, если матрица — действительно подходящая структура данных, а, например, не (key, key)->value хранилище на основе дерева, которое в случае персистентности достаточно эффективно. Плюс оптимизация, в коде x_new = x_old + 1 x_new могут занимать и одну область памяти, но это будет разруливать компилятор, избавив программиста от мыслей о состоянии.

konsoletyper 19 апреля 2012 в 13:45 +1

Ой, да многие структуры данных имеют свою функциональную реализацию, эффективно работающую в условиях, когда необходимо получать новую копию. Организуются они так, чтобы любое изменение требовало как можно меньше копирований уже имеющихся данных.

kingpin 19 апреля 2012 в 16:42 0

Статья в тему на Wikipedia. В конце дана ссылка на книгу Okasaki; в ней должно быть много полезного по поводу неизменяемых структур данных и их практическому применению.

MagicWolf 19 апреля 2012 в 18:10 0

>... функциональную реализацию, эффективно работающую...

Относительно эффективно.

По моим экспериментам с Clojure — для бизнес логики в самый раз, а вот число-дробилки (number crunching) быстрее выполняются в императивном стиле.

НЛО прилетело и опубликовало эту надпись здесь

soomrack 19 апреля 2012 в 13:20 +1

А может быть потому что простые вещи надо решать просто? Если, например, я пишу библиотеку для линейной алгебры, то зачем мне ФП? Лучше я реализую все алгоритмы с учетом функционирования аппаратной части и привяжу их к структурам данных, это ООП.

НЛО прилетело и опубликовало эту надпись здесь

НЛО прилетело и опубликовало эту надпись здесь

soomrack 19 апреля 2012 в 14:23 +1

А есть ли в ФП разделение на Алгоритмы и Структуры данных? Особенно если для с одной структурой мы хотим работать на разных уровнях абстракции? Те же матрицы — порой нам нужно работать с отдельными элементами, а порой со все матрицей в целом. Как это будет реализовано?

retran 19 апреля 2012 в 16:06 0

В ФП есть туплы и списки. А из них можно построить что угодно.

megalol 20 апреля 2012 в 00:17 0

Не зачем. Только библиотеки для линейной алгебры уже написаны, на них потрачены человеко-годы и нехрен велосипедить. А где использовать библиотеку уже большей частью все равно.

> Лучше я реализую все алгоритмы с учетом функционирования аппаратной части и привяжу их к структурам данных, это ООП.

ООП — это наследование, которое в случае линейной алгебры нафиг не нужно там ничего, кроме голой структурщины.

Scratch 19 апреля 2012 в 12:51 +15

Первая, имхо, человеческая статья по ФП, теперь я его не так сильно боюсь)

to_climb 19 апреля 2012 в 15:00 +5

Хотелось бы продолжения в том же ключе.

GnaeusPompeius 19 апреля 2012 в 12:59 0

```
String reverse(String arg) {
    if(arg.length == 0) {
        return arg;
    }
    else {
        return reverse(arg.substring(1, arg.length)) + arg.substring(0, 1);
    }
}
```

«бесконечная» рекурсия

TheShock 19 апреля 2012 в 13:02 +1

Аргументируйте.

mishadoff 19 апреля 2012 в 13:13 0

Наверное это шутка, раз бесконечная в кавычках

GnaeusPompeius 19 апреля 2012 в 13:14 +1

Пардоньте, неправильно распарсил скобки :)

НЛО прилетело и опубликовало эту надпись здесь

worldmind 13 сентября 2012 в 16:24 0

Ну да, и мне кажется бесконечная, т.к. мы всегда возвращаем строку той же длины (только с переставленным символом), то условие arg.length == 0 никогда не выполнится, а значит выхода нет

worldmind 13 сентября 2012 в 16:35 0

а, понял

Illorian 19 апреля 2012 в 13:12 +1

Я считаю героями тех, кто может до конца за раз прочитать таки статьи.

 **inossidabile** 19 апреля 2012 в 13:18 +17

А по-моему эта статья как-раз написано настолько хорошо, что глаз не оторвать. Возможно потому, что я искренне люблю ФП. Но и оригинал, и перевод, читаются на ура.

 **Seekeer** 19 апреля 2012 в 18:08 +2

Я с ФП не знаком, но статью прочитал запоем, наверно сказалось мат образование)
Правда, некоторые вещи так до конца не осознал.

 **konsoletyper** 19 апреля 2012 в 13:41 +7

В статье много наврано в пользу «прекрасного ФП».

> Unit тестирование

В реальности состояние есть (база данных, интерфейс пользователя), просто в этом случае ФП-программа будет выступать в качестве функции, которая берёт состояние мира в прошлой итерации (например, секунду назад) и возвращать новое состояние мира. Так что все зависимости сохранятся, просто их надо будет тащить в аргументы.

> Отладка

См. выше. Функция, преобразующая мир, может выполнить 1000 итераций, а на 1001-й отвалиться. Кстати, по моему горькому опыту, как раз-таки функциональные программы тяжелее отлаживать.

> Многопоточность

Отлично, вот только многопоточность в этом случае будет эквивалентна двум функциям, которые получают из 1000-й итерации 1001-ю. И при этом каждая получит свой кусок мира. Так вот этим функциям надо будет правильно разобрать мир на кусочки, преобразовать и собрать заново. А обычные вычисления, которые не трогают общего состояния, и на java прекрасно параллелятся.

> Развёртывание по горячему

Аналогично, если новая версия функции окажется несовместимой со старым состоянием мира, то мы новое состояние мира никак не получим, не перезапустив этот самый мир с нулевой итерации

> Доказательные вычисления и оптимизация

Это всё можно прекрасно делать с императивными языками. И делается. Тем более, что по сути, императивные программы преобразуются в граф, haskell тоже свои программы компилирует в граф редукции. Но это только haskell, потому что он академический и чисто ленивый, а вот многие другие ФЯ перед оптимизацией и генерацией машинного кода получают представление, слабо отличающееся от того же байткода Java.

> Функции высшего порядка

Этим давно уже никого не удивишь в мире императивного программирования. Благо, есть Python, Ruby и C#. Java тоже подтягивается.

> Ленивые вычисления

Это не является прерогативой одних только ФЯ. Во-первых, далеко не все ФЯ ленивые. Во-вторых, организовать ленивость в ИЯ при необходимости несложно.

> Pattern matching

А что мешает делать PM в императивных языках?

> Замыкания

Этим сейчас тоже никого не удивишь. Даже в Java есть довольно давно

retran 19 апреля 2012 в 16:09 +4

> В реальности состояние есть (база данных, интерфейс пользователя)

БД и интерфейс в юнит-тестах? Что-что, простите?

>... много всего с упоминанием слова «мир»...

SiCP в руки и учиться декомпозиции задачи.

konsoletyper 19 апреля 2012 в 16:25 +2

> БД и интерфейс в юнит-тестах? Что-что, простите?

Отлично, можно сказать «автоматические тесты», чтобы быть формально корректнее. Ну вот что делать, если код зависит от БД? Конечно, к БД обращаемся через интерфейс и в тестах даём тестируемому функционалу заглушки и mock'и. Но всё равно, даже такая непрямая зависимость от БД порождает зависимость от состояния.

> SiCP в руки и учиться декомпозиции задачи.

SiCP читали, декомпозицию умеем, монаду IO знаем. И чо? Как не было серебряной пули, магическим образом решающей обозначенные в статье проблемы, так и нет. ФЯ лучше ИЯ только в определённом круге задач (а в определённом вообще позорно сливает), хоть обчитайся SiCP.

retran 19 апреля 2012 в 16:48 +2

Корректнее тогда уж «интеграционные». Про каковые в статье ни слова.
А там где есть I/O, у вас есть соответствующая монада. Хотя я придерживаюсь мысли, что идеальный подход — гибридный.

Далее по пунктам:

Отладка

А вот на мой взгляд отладка проще. Тупо потому что flow сильно меньше зависит от состояния переменных и меньше вариантов, которые необходимо проверить.

Многопоточность

Map/reduce. Да, да, в императивных тоже можно. А откуда оно пришло в ГИБРИДНЫЕ языки ruby, python, C№ итд?

Функции высшего порядка

Ленивые вычисления

Pattern matching

Замыкания

И откуда оно в гибридных языках?

Развёртывание по горячему
Что-то мешает преобразовать? Каррирование на что?

konsoletyper 19 апреля 2012 в 17:03 0

> Корректнее тогда уж «интеграционные». Про каковые в статье ни слова.

Ну если говорить про чистые юниты, то они и в ИЯ не вызывают проблем. Интеграционные тесты тоже можно автоматизировать, да. Потому я про автоматические тесты сказал.

> А там где есть I/O, у вас есть соответствующая монада. Хотя я придерживаюсь мысли, что идеальный подход — гибридный.

А можно и без монад, если это не Haskell, а гибридный язык. Монада — это костыль для чистых ленивых ФЯ для обработки состояний. Такой же, какие бывают в ИЯ для организации ленивости и т.п.

> Функции высшего порядка
> Ленивые вычисления
> Pattern matching
> Замыкания

Учитывая, что оно прекрасно прижилось в ИЯ, стоит ли эти фичи называть монопольными фичами ФЯ? По сути, что такое ФЯ? Это язык, где мы напрямую не оперируем состоянием? Где функции не дают побочных эффектов?

> Развёртывание по горячему
> Что-то мешает преобразовать? Каррирование на что?

Тут без конкретики сложно о чём-то говорить. Например, вот есть сервлет на Java. При старте в контейнере у него ORM обходит классы модели и компилирует шаблоны. На это тратится некоторое время. Кто мешает контейнеру быть чуть умнее и запускать второе окружение для сервлета, а когда он полностью запустится, выключать первое окружение со старым сервлетом и все запросы начать направлять ко второму? Как бы проблема решилась, если бы мы использовали haskell?

 **retran** 19 апреля 2012 в 17:20 0

Согласен, что монады — костыль.

> Учитывая, что оно прекрасно прижилось в ИЯ, стоит ли эти фичи называть монопольными фичами ФЯ? > По сути, что такое ФЯ? Это язык, где мы напрямую не оперируем состоянием? Где функции не дают
> побочных эффектов?

В первую очередь — функции высших порядков и декларативный стиль программирования. ФЯ — это не столько языки, сколько определенный стиль программирования и мышления.

Про горячие развертывание:
Честно говоря не знаю как в хаскелле, но в эрланге примерно так и происходит. Только это все инкапсулировано от разработчика и происходит «по волшебству».

 **konsoletyper** 19 апреля 2012 в 17:47 +2

> функции высших порядков

В Python, Ruby и C# есть ФВП. Вопрос: почему мы Haskell, Erlang или Ocaml, где они тоже есть, считаем функциональными, а другие языки — не считаем? Уж не из-за декларативного ли стиля? Я так скажу, что благодря fluent-интерфейсу на C# определённые вещи пишут очень и очень декларативно.

> декларативный стиль программирования

Я так понимаю, что декларативный стиль от императивного отличаются тем, что в первом случае мы сказали «хочу в Тамбов» и нас интерпретатор повёз, а в императивном мы рулим-рулим, и внезапно туда попадаем. Я понимаю, что первое удобнее и быстрее, однако ФЯ такого не дают. Я пишу быструю сортировку на haskell-е в одну строчку и он мне сортирует мееедленно. Если хочется быстро, мне надо писать какое-то там слияние. Т.е. я рулю сам. Что я делаю не так? Нет, ну т.е. я прекрасно понимаю, чем ИЯ от ФЯ отличается, но даёт ли это какое-то реальное преимущество одному перед другим?

> в эрланге примерно так и происходит. Только это все инкапсулировано от разработчика и происходит «по волшебству».

Отлично. Кто указанное поведение мешает реализовать в контейнерах сервлетов? При этом оно тоже будет совсем инкапсулировано от разработчика.

retran 19 апреля 2012 в 17:52 +1

> Я так понимаю, что декларативный стиль от императивного отличаются тем, что в первом случае мы сказали «хочу в Тамбов»...

Нет. Такое понимание приходит от кривоватых определений в статьях.
Декларативный стиль, это когда вы задаете набор определений от самый простейших (в ФЯ определения задаются в виде функций, поэтому они и ФЯ), а интерпретатор их уже рекурсивно крутит.
Т. е. в вашем примере:
«Тамбов — это город»
«Поезда ездят из города в город»
«Я сижу в поезде не в Тамбове, а поезд едет в Тамбов.»
итд.

VoidEx 19 апреля 2012 в 20:34 +1

IO тоже можно без монад. Монада к IO имеет весьма посредственное отношение и вовсе не костыль, коль уж имеются монады Exception, Maybe, State, Async (async/await), Cont (call/cc) и другие.
Эдак можно сказать, что async/await — костыль для языков, где нет нормальных монад.

Хотя можно придерживаться т.з., что объекты — замыкания для бедных, а замыкания — объекты для бедных.

konsoletyper 19 апреля 2012 в 21:35 0

Тяжело будет делать IO без монад вообще. Монада != спец. синтаксис для неё в haskell. И про остальные монады я знаю. Я не говорил, что монада как таковая костыль для haskell. Конкретно монада IO + сахар в виде do — это и есть костыль для состояния.

VoidEx 19 апреля 2012 в 22:13 0

Сахар в виде do — он для всех монад.

Ну, а Async-monad — костыль для асинхронный действий?
А Cont-monad, надо полагать, костыль для call/cc?

В чём костыльность-то проявляется?

konsoletyper 19 апреля 2012 в 23:36 0

Вот на примере Java. Там не сделали нормального вывода типов, потому надо писать:

```
Map<String, Integer> someMap = new HashMap<String, Integer>();
```

Добрые люди пишут различные библиотеки, которые позволяют сделать покороче:

```
Map<String, Integer> someMap = Collections.newMap();
```

Это костыль. Теперь насчёт Haskell. Его природа обделила императивными конструкциями. Поэтому работу с состояниями надо делать в монаде IO. Вроде бы это можно делать, но возникает ряд сложностей. Например, новичку надо разобраться, почему в одних случаях можно написать

```
do
   a <- someFoo
```

а в других

```
do
   a <- return someFoo
```

Или почему в одних случаях нужен map, а в других — mapM. Ну и т.п. В Java, чтобы вызвать императивный

метод или организовать императивный цикл, не нужно знать тонкостей реализации JVM, потому что в Java императивность есть нативно. А вот в Haskell — костыль.

VoidEx 20 апреля 2012 в 00:00 +1

> новичку надо разобраться
То, что новичку что-то там не понятно, не тянет на аргумент о костыльности в системе.
Может, и теория групп в математике тогда костыли?

map от mapM отличаются не только тем, что там монада, а и тем, что для map выполняются функторные законы, для mapM — нет.
Тот же mapM можно написать как минимум двумя способами.

В хаскеле не нужно знать «тонкости», надо понимать, что такое чистота. Новичку и впрямь непривычно, но на костыльность это не указывает.

konsoletyper 20 апреля 2012 в 07:32 0

Отлично. Тогда договоримся, что мы будем считать критерием костыльности?

VoidEx 20 апреля 2012 в 13:02 0

Меня устраивает это: lurkmore.to/%D0%BA%D0%BE%D1%81%D1%82%D1%8B%D0%BB%D1%8C

VoidEx 20 апреля 2012 в 13:04 0

Случайно отправил, недописав.

В случае языков ситуация, когда недостающая функциональность обходится другими средствами языка, при этом крайне неудобно. Как лямбды в C++03, например (boost::bind + boost::lambda).

Так вот чистота и ленивость — это фундаментально в Haskell, и сделано намеренно. И различие между IO/State/whatever и чистыми функциями тоже сделано намеренно. Это меняет подход, это неудобно новичкам, но это не костыль.

У сколь-нибудь опытных программистов на Haskell с этим проблем нет.

VoidEx 20 апреля 2012 в 19:43 0

Задам встречный риторический вопрос.

В Haskell STM (Software Transactional Memory) реализовано в виде библиотеки. Благодаря тем самым «костыльным» монадам для состояния, у нас есть гарантия того, что всё STM изолировано и мы не можем внутри обращаться к «левым» ссылкам.
Как дать такие гарантии там, где потенциально всё — IO?

Также есть монада ST, мы можем локально заводить мутабельные переменные, но так как они изолированы, итоговое вычисление гарантированно чистое. Оборачиваем это в runST и получаем чистое значение, реализованное императивным способом. Это аналогично pure из D, но без дополнительного ключевого слова и какого-то специального анализа компилятора.
Вопрос тот же, как это сделать в языке, где нет таких «костылей»? В D пришлось ввести новое ключевое слово.

konsoletyper 21 апреля 2012 в 09:49 0

В Java можно менять байт-код как душе угодно. На этом и основаны некоторые библиотеки, реализующие STM в Java.

VoidEx 21 апреля 2012 в 11:41 0

Если внутри STM-кода я меняю внешнюю переменную, то что будет?

konsoletyper 21 апреля 2012 в 22:13 0

Ой, ну что вы въелись так? Не приходилось мне использовать STM, потому тонкостей я не знаю. Ну не мой это конёк. Могу предположить, что «а если я себе выстрелю в ногу» — примерно из той же оперы вопрос. В крайнем случае, всегда можно написать утилиту, которая проанализирует байт-код и поругается на места, где можно поломать STM, и даже свой SuppressWarning для неё предусмотреть, мол, я знаю, чего хочу. Причём, это может быть даже не утилита, а плагин к одному из множества анализаторов кода для Java. Или плагин к встроенному анализатору кода IDE (для Netbeans и IDEA такое точно можно сделать, для Eclipse не приходилось, но тоже, может быть, возможно).

Лучше уж я задам вопрос по тому, что я знаю. Вот если есть у нас какая-то логика на веб-сервере, и её нужно реализовать один-в-один в браузере на JS (например, логика валидации или парсер языка разметки для превью в блоге), то как такое могут решить средства Haskell (не считая написания компилятора самого Haskell)?

cblp 8 декабря 2014 в 09:17 0

Уже есть несколько компиляторов Haskell в JavaScript разной степени удачности. Я на них пока вплотную не смотрел, подробнее рассказать не могу, но вот вам ссылка www.haskell.org/haskellwiki/The_JavaScript_Problem#Haskell_-.3E_JS

megalol 20 апреля 2012 в 00:24 +1

> SiCP читали, декомпозицию умеем, монаду IO знаем. И чо?

Чо чо, «все в IO», — это не ФП, а то, во что превращается код, втупую перенесенный с императивного языка. ФП — это выкинуть из IO все, что можно, оставив легко отлаживаемый ссылочно-прозрачный минимум.

konsoletyper 20 апреля 2012 в 07:36 0

Отлично. Что делать, если состояние всё-таки есть? Как я уже говорил в комментах к другому топику о функциональных языка, я с этим столкнулся при написании компилятора. Ну вот есть множество входных символов, положение в тексте (номер строки/номер столбца), накопленные ошибки. Если таскать всё это а аргументах — можно убиться. И вообще, я не пишу о том, что надо везде использовать IO. Я просто хотел показать, как все перечисленные достоинства ФЯ исчезают при встрече с реальным миром, в котором есть состояние.

VoidEx 20 апреля 2012 в 13:09 +1

Странно, Agda, GHC и куча других компиляторов написано на Haskell как раз потому, что он для этого удобен.
Я пишу на нём уже несколько лет, писал WebSocket-сервер, mp3tags-декодер (ибо текущие реализации на тот момент на любом языке многие не поддерживали даже сжатия, не говоря о шифровании), и почему-то пока не столкнулся с ситуацией, где «все перечисленные достоинства» исчезали бы.

burjui 19 апреля 2012 в 17:01 0

Когда критикуешь что-то, важно вовремя остановиться, чтобы не скатиться в критику всего подряд, даже если критиковать нечего.

> Это всё можно прекрасно делать с императивными языками

> Этим давно уже никого не удивишь

> А что мешает делать PM в императивных языках?

> Этим сейчас тоже никого не удивишь

burjui 19 апреля 2012 в 17:06 +1

Прошу прощения, отправил раньше времени.
Так вот, цель статьи не в том, чтобы кого-то удивлять «уникальными фичами», а в том, чтобы дать представление об основных инструментах ФП. И если всё это «можно», то почему в императивных языках всё это так непопулярно? Не потому ли, что в ФП такие лучше интегрируются и смотрятся естественнее?

konsoletyper 19 апреля 2012 в 17:10 0

Ну насчёт лучше интегрируются и смотрятся естественнее — это вопрос спорный. А в каких императивных языках это непопулярно?

RiderSx 19 апреля 2012 в 14:17 +2

Дочитал. Спасибище огромное, как уже заметили выше — очень годная статья для новичков в ФП :) И да, читал вдумчиво и не торопясь выпив кружки три кофе, без малого полтора часа заняло :)

ENargit 19 апреля 2012 в 15:13 +1

Занимательная статья и хороший перевод. Но с отождествлением каррирования и шаблона Адаптер автор, кажется, переборщил. Каррирование — только один из сценариев использования адаптера, а ведь есть и другие.

Funcraft 19 апреля 2012 в 15:57 −1

Хорошая, интересная статья.

Однако, на мой взгляд, автор давно (или вообще?) не работал с web-яп в духе Python, Ruby, PHP, так как в них многое, что в статье описывается как «невозможное в императивных языках» является вполне себе возможным и даже не считается магией.

havelock 19 апреля 2012 в 16:12 0

>Python, Ruby

их никак нельзя называть «web-яп» и поставить в один ряд с PHP. Даже Ruby, который для меня привычнее всего воспринимается как "...on Rails" :) Это general-purpose ЯП, и питон в огромном числе своих применений не имеет к вебу никакого отношения.

Funcraft 19 апреля 2012 в 16:40 +1

Хм, PHP вообще-то тоже без вэба можно использовать, но вы же поняли о чём я?

avesus 19 апреля 2012 в 16:05 0

Большое спасибо за перевод. Оригинал бы тоже осилил, но нашёл бы лет через 100 — хабр как-то чаще читается :-)
Очень надеялся здесь же прочитать про монады, но в конце увидел жуткий облом.
Скажите, автор уже написал на английском, или это «свежачок»? Потому что мне кажется, что понимание монад — это именно то, чего мне не хватает, чтобы представить, что все эти чисто функциональные программы вообще могут работать. И почему без монад вообще никак?

Seekeer 19 апреля 2012 в 18:12 0

Там же вначале написано:
«Понедельник, 19 июня, 2006»
Так что, думаю, автор уже написал:)

avesus 19 апреля 2012 в 22:53 +1

Смотрел другие его публикации — нигде нет намёка на продолжение. Отправился в самостоятельное копание за что получил по шапке по причине непроизводственного использования рабочего времени :-)

avesus 19 апреля 2012 в 16:36 0

Наконец-то в википедии нашёл замечание, способное пролить свет на суть монад:
«Although a function cannot directly cause a side effect, it can construct a value *describing* a desired side effect, that the caller should apply at a convenient time.»

Если я правильно понял, монады используются для того, чтобы, вызвав какую-ту функцию с желательным сайд-эффектом, они

вернули *описание* этого сайд-эффекта, которое можно куда-то применить. Самый непонятный момент — «применить». Реально ли написать ОС на чисто функциональном языке?

404 **amarao** 19 апреля 2012 в 19:41 +1

Я что-то слышал про ковыряние ОСи на Хаскелле. Основная проблема — чем ниже уровень, тем больше возникает императивного кода с острейшими требованиями по скорости исполнения.

Например, верхние половинки прерываний требуют быть настолько быстрыми, насколько возможно. Каждый такт, потерянный в этой половинке рушит абстракции всех остальных систем.

Наверное, если попытаться спроектировать железо так, чтобы можно было комфортно обрабатывать прерывания, управлять скедулером и т.д., получится лучше...

 **avesus** 19 апреля 2012 в 22:55 0

ИМХО без специально заточенного под это дело железа смысла писать ось нет. У меня просто другое когнитивное затруднение — может быть, чтобы такая система в принципе могла быть написана на чисто функциональном языке, она требует специально заточенный под неё *мир*? Чисто функциональный... :-)

404 **amarao** 19 апреля 2012 в 23:24 0

Насчёт мира не знаю. Интерфейсы — может быть.

НЛО прилетело и опубликовало эту надпись здесь

 **avesus** 20 апреля 2012 в 14:44 +2

Всё, перехожу на Xen!

404 **amarao** 20 апреля 2012 в 16:36 0

Ну это просто феерия какая-то. Разумеется, паравиртуализированная система может быть написана на чём угодно. Причина очень простая — паравиртуализированный домен физически не имеет права исполнять привилегированные инструкции. За него всё делает dom0 (а там линукс, ага, со своими мерзкими императивными драйверами к чему попало) и xen (который разруливает прерывания и управляет памятью).

Это всё равно, что показывать куличик в песочнице и говорить про возможность строительства небоскрёбов на песчаном грунте.

НЛО прилетело и опубликовало эту надпись здесь

404 **amarao** 20 апреля 2012 в 16:44 +1

Я про это и говорю. Паравиртуализированный домен (если в него не делегируется PCI-шина) находится от железа на большем расстоянии, чем userspace-программа обычного домена.

Все паравиртуализированные драйвера (blkfront, xen-netfront) проектировались так, чтобы всё самое сложное было в -back половинке, а front был бы примитивным и простым в реализации (для упрощения портирования ОС под Xen).

Другими словами, наличие чего-либо под xen означает лишь способность делать гипервызовы (которые от syscall отличаются лишь номером прерывания), да способность прочитать свою start page. Кроме этого, машина в домене ничем от обычного ELF-приложения не отличается.

 **megalol** 20 апреля 2012 в 01:50 +1

Это не суть монад, это частный случай State.
Возьмите код для maybe, list, writer и state (в таком порядке, сложность будет возрастать) и раскройте скобки, превратив do-нотацию во вложенные bind'ы, а потом раскрыв бинды в вызовы функций. Там все видно будет, что общее, а что частное. И каким образом state (и IO) тащит состояние.

spmbt 19 апреля 2012 в 18:59 +2

Мысли в процессе чтения:
1. Pattern matching — широкое использование в CSS, XSLT.

2. Функция-наследник — это коллбеки в современных языках и 3-адресная система команд в машинных языках.

3. Есть какое-то противоречивое противопоставление возврата значений и передачи результата. Передача результата — это возврат. Как можно говорить, что первое не нужно, потому что есть второе?

404 **amarao** 19 апреля 2012 в 19:38 0

Это всё всегда красиво выглядит, пока речь не заходит о том, что нужно работать с байтами и битами. Что нужно СНАЧАЛА записать в порт 0xFFFE байт 0xF3, _ПОТОМ_ через 0.3-0.35нс прочитать из порта 0xFFE значение, _ПОТОМ_ начать через указанное число НС и читать массив значений с частотой 10000000 шт в секунду из порта 0xFFFA с допуском не более ± 0.1мс.

Leeb 19 апреля 2012 в 23:17 0

Это все от того, что существующее железо императивно. Все-таки ФП тянет абстракции «вверх» а не «вниз».

404 **amarao** 19 апреля 2012 в 23:26 0

Попробуйте описать простенький интерфейс к шаговому двигателю без интеллекта в функциональных терминах.

Дано: на ногу поступает напряжение, двигатель делает поворот на 1 градус. Даётся напряжение на другую ногу — в другую сторону. Длительность импульса — не менее 10мс, можно дольше, но не сильно, ибо греется (например, не более 50мс). Заметим, никакой логики и интеллекта. На ноге А 1 — ползём влево на шаг. На Ноге Б — вправо на шаг. Для двух шагов нужно повторить цикл появления исчезновения напряжения с паузой не менее 3мс.

Попробуйте предложить этому функциональную обвёртку. Кстати, хорошая задача для любителей ФЯП.

megalol 20 апреля 2012 в 00:40 +2

FRP — список (время, действие) превращается в список (время, напряжение на ногах). Этот список хавается единственной грязной функцией, которая просто устанавливает нужные напряжения в нужный момент времени. Дальше для любого типа двигателя нужно просто представить набор функций, который мало будет отличаться от таблички в документации. Хороший код на C будет выглядить примерно так же, но этому еще и научиться нужно, скорее всего этот код будет одноразово захардкожен.

404 **amarao** 20 апреля 2012 в 00:42 0

Стоп. Мы с вами в реальном мире живём. Что значит «в нужный момент времени»? Какого времени? wallclock или таймера? Кто таймеры обрабатывает и отсчитывает?

megalol 20 апреля 2012 в 00:49 0

Системная sleep?

404 **amarao** 20 апреля 2012 в 00:53 0

Мне кажется, или мы обсуждаем как это всё писать на ФЯПе?

Кроме того, 10мс вы получите очень неточно (если мы про современное железо и обычный sleep), куда точнее использовать высокоточные таймеры.... Или считать тики процессора (спинлоки). Я с трудом понимаю, как это всё можно уложить в парадигму красивости.

... А главное, чем ближе к железу, тем меньше там алгоритмов высокого уровня и больше именно таких «прочитать, поспать, записать, вернуть, изменить бит» и т.д.

VoidEx 20 апреля 2012 в 01:03 +2

www.cs.yale.edu/c2/images/uploads/AudioProc-TR.pdf

Смотрите сразу страницу 24 (картинка) и 25 (код).
Оцените декларативность, схема почти напрямую переписана.

megalol 20 апреля 2012 в 01:30 +3

> Мне кажется, или мы обсуждаем как это всё писать на ФЯПе?

Я объяснил, как это писать на ФЯПе. Хотите считать тики — считайте в Sleep тики, или используйте PTOC, что бы вы на C делали, то и тут делайте в общем. Это будет одной грязной функцией из двух. Вторая — та, которая состояние на ногах меняет.

> А главное, чем ближе к железу, тем меньше там алгоритмов высокого уровня и больше именно таких «прочитать, поспать, записать, вернуть, изменить бит» и т.д.

Смотря к какому железу. Те же видеокарты намного проще в терминах map и reduce программировать. С VHDL тоже не все так неоднозначно, там по сути два языка, императивный и декларативный, и декларативно обычно проще описывать. Хотя я не особый знаток VHDL.
Если речь о контроллерах и C, то скрипты типа «прочитать, поспать, записать, вернуть, изменить бит» низкоуровневы, но не особо грязны. Есть входное состояние, выходное состояние, скрипт между ними. Шаговый ли это двигатель или протокол обмена, набор реально низкоуровневых примитивов мал. Для шагового двигателя их два, например.

voidlizard 20 апреля 2012 в 09:41 +2

Запись в порты — это IO в любом случае. Можно сделать на продолжениях.
В нечистых ФЯ реализуется ровно так же как в императивных. Т.е проблемы в принципе нет, как и особой «задачи для любителей ФП»

ShadowPrince 19 апреля 2012 в 21:34 −4

Положил статью в папку «прочитать»

ShadowPrince 21 апреля 2012 в 02:46 +2

Оказывается кроме «статью_не_читай@комментарий_пиши», есть еще «комментарии_не_читай@комментарий_пиши».

nick4fake 19 апреля 2012 в 22:12 0

<< действительно работающей машиной Тьюринга
Вот прям так? Машина Тьюринга в реальном мире? С бесконечной лентой? Основа машины тьюринга именно эта бесконечность.

cblp 8 февраля 2015 в 18:09 0

В реальном мире можно использовать:

«гарантированную бесконечность» — лента конечная, но настолько большая, что мы уверены, что те программы, что мы запускаем, не доберутся до её края.

«потенциальную бесконечность» — когда головка доходит до конца ленты, машина останавливается, специально обученный человек наращивает ленту, машина продолжает работать.

incogn1too 19 апреля 2012 в 22:26 +1

Спасибо за статью. Историческая часть была интересной, но примеры на Джаве смотрелись странно.

sheknitrtch 19 апреля 2012 в 22:34 +1

Автор оригинальной статьи ставил себе целью объяснить ФП для любителей императивных языков программирования. Видимо он решил, что Java будет понятней большинству.

ShadowPrince 19 апреля 2012 в 23:03 +1

Ява запутывает и очень хорошо.
Немного врубился только после tryhaskell.org/

eyeofhell 19 апреля 2012 в 23:27 +1

> Ленивые языки позволяют создавать бесконечные структуры данных, создание которых в строгих языках гораздо сложнее [пер. — только не в Python].

Python сложно отнести к «строгим» языкам — он мультипарадигменный и содержит достаточно много функциональных элементов. В частности, генераторы бесконечных последовательностей с помощью ключивого слова «yield» :).

sheknitrtch 20 апреля 2012 в 10:28 0

Я как раз и вспомнил про `yeild` и генераторы, когда увидел «бесконечные структуры данных».

kuchumovn 20 апреля 2012 в 00:09 +1

очень крутая статья, давно здесь таких не было.
ловите плюс.

Alex_Tri 20 апреля 2012 в 06:56 0

Так, про ФП на понятном языке прочитал, спасибо, теперь надо найти что-нибудь подобное про лямбда-исчисление.

НЛО прилетело и опубликовало эту надпись здесь

Alex_Tri 20 апреля 2012 в 15:29 0

Именно 'Real World Haskell' я сейчас и читаю. А теория мне скоро понадобится для реферата.

naryl 20 апреля 2012 в 07:45 0

> В императивном мире это невозможно [пер. в Smalltalk-е очень даже возможно]. Представьте себе выгрузку Java класса на лету и перезагрузка новой версии. Если бы мы так сделали, то все экземпляры класса стали бы нерабочими, потому что потерялось бы состояние, которое они хранили.

В Common Lisp тоже возможно. С перезагрузкой определений классов. (Нет, он не функциональный, дальше даже о этом сказано)

qnub 20 апреля 2012 в 07:51 +1

750 добавило в избранное, рейтинг +150... Вы чего человеку в карму не плюсуете?!

charon 2 мая 2012 в 18:40 0

да просто потому что в избранное может добавить кто угодно, а оценки доступны очень мало кому

ncix 20 апреля 2012 в 10:09 0

Отличная статья, теперь мне как заядлому ООПшнику (C++, Delphi, C#) функциональщина не кажется таким WTFом как раньше.

Впрочем, как всегда, одни плюсы и никаких серьезных минусов парадигмы не указано. А так не бывает. Это настораживает.

Опять же на вопрос почему функциональщина не шагает по планете, приводятся примеры в общем-то специфического телекоммуникационного софта. А как насчет более традиционных приложений? Корпоративный сегмент например? Работа с БД?

НЛО прилетело и опубликовало эту надпись здесь

 **ncix** 20 апреля 2012 в 14:00 0

Спасибо за обзор инструментов, но меня больше интересует опыт. Опыт разработки корпоративных приложений с использованием ФП. Какие плюсы/минусы.

Кстати отдельный вопрос, как насчет GUI? Десктопные GUI-приложения чисто на ФП пишут ли?

НЛО прилетело и опубликовало эту надпись здесь

 **sheknitrtch** 20 апреля 2012 в 10:33 +4

Спасибо за тёплый приём. Были интересно читать комментарии.
Удивительное дело, Я почти неделю занимался переводом. В сумме четыре раза перечитывал весь пост с включённой проверкой орфографии, и всё равно в текст проникли опечатки и ошибки. Спасибо всем, кто писал мне на почту замечания. По состоянию на момент написания этого комментария в тексте было исправлено 10 опечаток. Это так, для статистики :)

 **DZhon** 28 апреля 2012 в 14:36 0

Вы выбрали прекрасную статью для перевода!

 **sergeypid** 20 апреля 2012 в 10:35 0

А посоветуйте книгу или курс для изучения Эрланга

 **Leeb** 20 апреля 2012 в 11:11 +8

Learn You Some Erlang for Great Good! или Erlang Programming для начала и Erlang and OTP in Action для продолжения.

 **G0ran** 20 апреля 2012 в 10:36 +1

Спасибо за отличный, качественный перевод, полностью осилил и многие вещи стали понятные! Очень хочется продолжея в таком же ключе и такого же качества!

 **ZyL** 26 апреля 2012 в 10:05 0

Я никогда не работал с фонкциональными языками, но когда я встречаю что-то, чего не знаю, я смотрю — а сколько людей с такими навыками требуется? Смотрю обычно на монстре в штатах — самый большой рынок.

Вот и сейчас — эрланг 28 ваканский по всем штатам, а хаскела вообще нет. Хотя я названия языков встречаю достаточно часто.

Т.е. получается, что все это интересно (статью прочитал с удовольствием), но на практике (почти) нигде не используется? Кто-нибудь из присутствующих сам разрабатывал реальные коммерческие проекты на этих языках (ну или слышал от друзей)?

 **worldmind** 13 сентября 2012 в 17:06 0

> В данной статье мы поработаем с продолжениями, а монады и однозначную типизацию отложим до следующего раза

следующий раз был/будет?

 **worldmind** 13 сентября 2012 в 17:07 0

чуть промахнулся

Jeer 16 августа 2016 в 20:26 0

Очень интересная статья. Прям завораживает, действительно раньше функциональное программирование пугало, а после этой статьи почитал про него еще. Сидя в своей уютной visual studio с уютным c#, разумеется, начал копать в сторону языка f#. И вот что я понял, поправьте меня, если ошибаюсь: являясь дотнетовским языком у f# остался лишь «лаконичный синтаксис». То есть, один из озвученных плюсов — что легко распараллеливать код, на самом деле, один и тот же код на разных языках, c# await/async и f#, со своим подходом, компилируется в один и тот же промежуточный IL. То есть какого-то выигрыша производительности нет и разница только в синтаксисе? Другими словами, f# не является чистым функциональным языком, поэтому в нём нет никаких других преимуществ или плюсов?

НЛО прилетело и опубликовало эту надпись здесь

solver 29 ноября 2016 в 00:30 0

Да что же вы все уперлись в эту параллелность.
Это вообще десятое дело, вопринимайте ее просто как приятный побочный бонус.
Вся эта функциональная чистота нужна исключительно для человека. Банально потому, что он не может оперировать 0 и 1 для составления сложных систем. Ему нужны абстракции для написания программ. Если бы программы писали роботы, то они прекрасно оперировали 0 и 1. И никогда не ошибались при этом)
Все эти навроты нужны только для того, чтобы программист ошибался меньше и легче понимал что написали другие программисты.
Поэтому абсолютно не важно, что c# await/async и f#, со своим подходом, компилируется в один и тот же промежуточный IL.
Начхать с высокой колокольни. Главное тут удобство и корректность для разработчика. А у F# этого больше чем у C#.

НЛО прилетело и опубликовало эту надпись здесь

solver 29 ноября 2016 в 12:23 0

> Почему Вас это так сильно раздражает?
Раздражает?))) Меня не раздражает, а удивляет дремучесть людей, которые из всего многообразия возможнотей, выбирают в качестве основного аргумента фактически побочный эффект. Очень приятный и полезный безусловно. Но все же не являющийся ключевой особенностью.

>Так почему об этом нельзя говорить?
Почему же нельзя говорить? Можно конечно. Только говорить надо корректно)
Выражения типа «на самом деле, один и тот же код на разных языках, c# await/async и f#, со своим подходом, компилируется в один и тот же промежуточный IL. То есть какого-то выигрыша производительности нет и разница только в синтаксисе? Другими словами, f# не является чистым функциональным языком, поэтому в нём нет никаких других преимуществ или плюсов?», просто показывают дремучесть человека, который не понимает о чем вообще идет речь.
И вот в таком ключе говорить лучше не надо. Потому что многих, прочитавших такие высказывания это введет в заблуждение. Они будут думать, что все преимущество того же f#, должно быть в каком-то особенном байткоде. А это очевидная чушь.
Повторюсь, вся эта функциональщина, все эти навороты, нужны исключительно человеку, а не компьютеру.

НЛО прилетело и опубликовало эту надпись здесь

sanchower 28 сентября 2017 в 15:48 0

Идеальная статья для pocket

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

Судьба предателя, угнавшего новейший МиГ-25 в Японию

+58 285k 86 1648

Кофе и чай удивительно полезны для здоровья

+17 76,9k 67 126

На что соглашается человек, когда разрешает все куки

+53 31,4k 83 201

Почему язык Go стал стандартом для DevOps-инженеров

+43 17,3k 67 109

Самоучкам здесь не место? Опрос о саморазвитии в IT

Опрос

Ваш аккаунт
Войти
Регистрация

Разделы
Публикации
Новости
Хабы
Компании
Пользователи
Песочница

Информация
Устройство сайта
Для авторов
Для компаний
Документы
Соглашение
Конфиденциальность

Услуги
Реклама
Тарифы
Контент
Семинары
Мегапроекты
Мерч

© 2006 – 2021 «Habr» Настройка языка О сайте Служба поддержки Мобильная версия