

Почему Haskell хороший, годный язык программирования

Влияние идей из функционального программирования

Функциональное программирование давно является источником новых идей, но до недавнего времени (конца 1990-х – начала 2000-х) языки ФП развивались отдельно от мейнстрима программирования. Но в последнее время наметилась тенденция проникновения идей из ФП в массы:

Сборка мусора: появилась в LISP (1960), стала мейнстримом в 1995 (Java)

Замыкания: появились в Scheme (1975), стали мейнстримом в 2000-е (C#, JavaScript)

Вывод типов (type inference): появился в ML (1979), стал мейнстримом в 2007 (C#, Scala)

Ленивая обработка (lazy evaluation): появилась в Miranda (1985), стала мейнстримом в 2000-е (LINQ)

Появляются также гибридные языки (смесь ООП и ФП), хотя пока они не столь популярны, как Java или C#: **Scala** и **F#**.

Функции высшего порядка

В настоящее время уже общепринято, что функции высшего порядка часто позволяют сократить и упростить запись алгоритмов обработки данных в коллекциях, причем из этих функций легко строить целые "конвейеры" обработки данных (см. LINQ).

Еще в 1990-е лямбда-выражения, замыкания, функции высшего порядка казались экзотикой из мира ФП; в 2000-е они уже считаются обычным инструментом mainstream-программиста.

Императивный код

```
out = new int[in.length];  
for (i = 0; i < in.length; i++) {  
    out[i] = in[i] * 2;  
}
```

Функциональный код

```
out = map (\x -> x * 2) in  
-- или, что то же самое:  
-- out = map (*2) in
```

Пример – история языка C#:

1.0 (2001) – традиционный ООП-язык

2.0 (2006) – добавлены анонимные делегаты (лямбда-выражения)

3.0 (2007) – LINQ (поддержка ФВП в языке и библиотеках),
локальный вывод типов переменных

Функции высшего порядка

По [оценкам](#) Тима Суини (ведущего разработчика Epic Games), циклы в императивных программах (на примере анализа кода Unreal Engine) являются:

- в 40% случаев – комбинация функций map и filter (выражаемая в Haskell через списочные выражения – list comprehensions)
- в 50% случаев – свертка (fold)

```
String str = "Hello World";  
  
String result = "";  
for (i = 0; i < str.length(); i++) {  
    if (str[i].isUpper())  
        result += str[i].toLowerCase();  
}
```

```
-- list comprehension  
result =  
    [toLowerCase c | c <- "Hello World", isUpper c]
```

```
sum = 0;  
for (i = 0; i < arr.length; i++) {  
    sum += arr[i];  
}
```

```
-- свертка  
sum = foldl (+) 0 arr
```

Вывод типов (type inference)

Одно из направлений критики языков со статической типизацией – их многословность (подчеркнуты явные объявления типов):

```
List<T> replicate(int n, T elem) {  
    List<T> list = new ArrayList<T>(n);  
    for (int i = 0; i < n; i++) {  
        list.add(elem);  
    }  
    return list;  
}
```

Вывод типов (type inference), оставляя статическую типизацию, позволяет не писать явные объявления типов – компилятор выводит их сам из контекста. Поэтому программы на Haskell могут вообще не содержать явных объявлений типов (как в динамических языках), при этом оставаясь при всех преимуществах статической типизации.

```
replicate 0 _ = []  
replicate n elem = elem : replicate (n - 1) elem
```

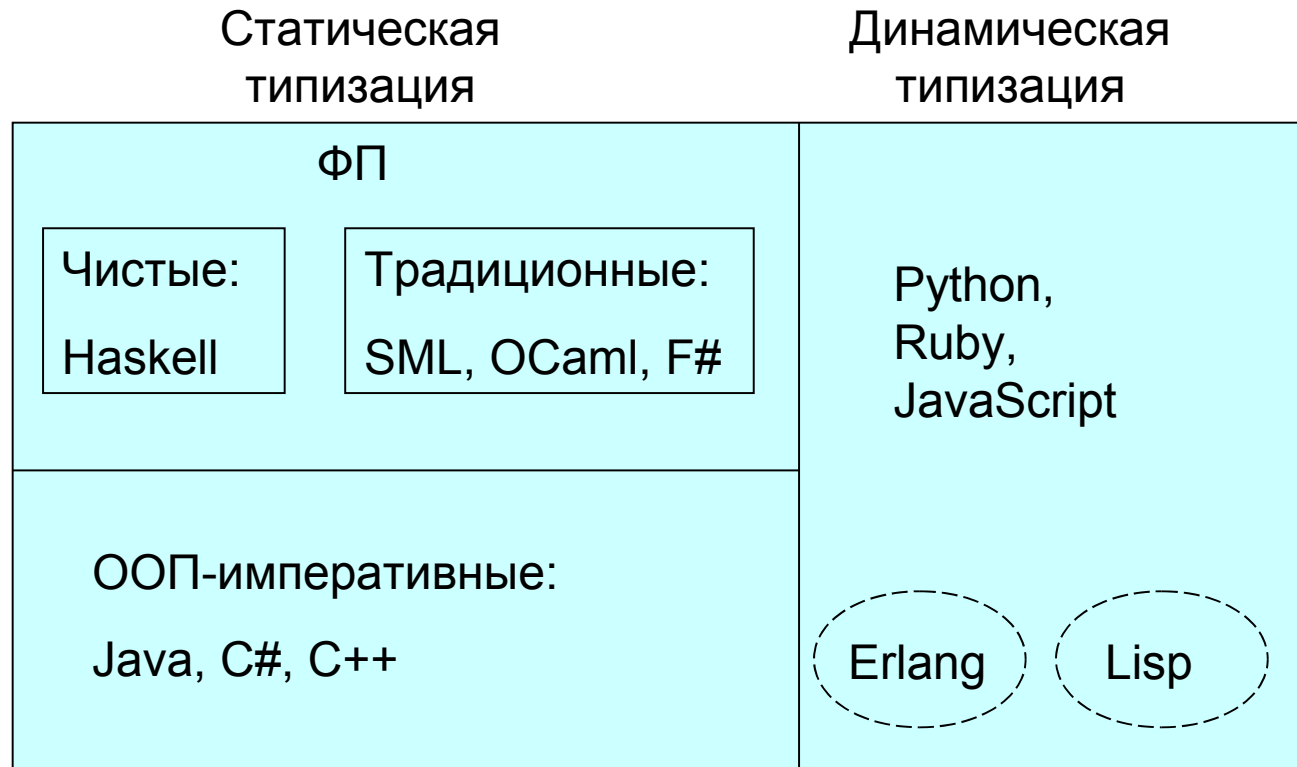
Языки без вывода типов



http://users.livejournal.com/_winnie/227010.html

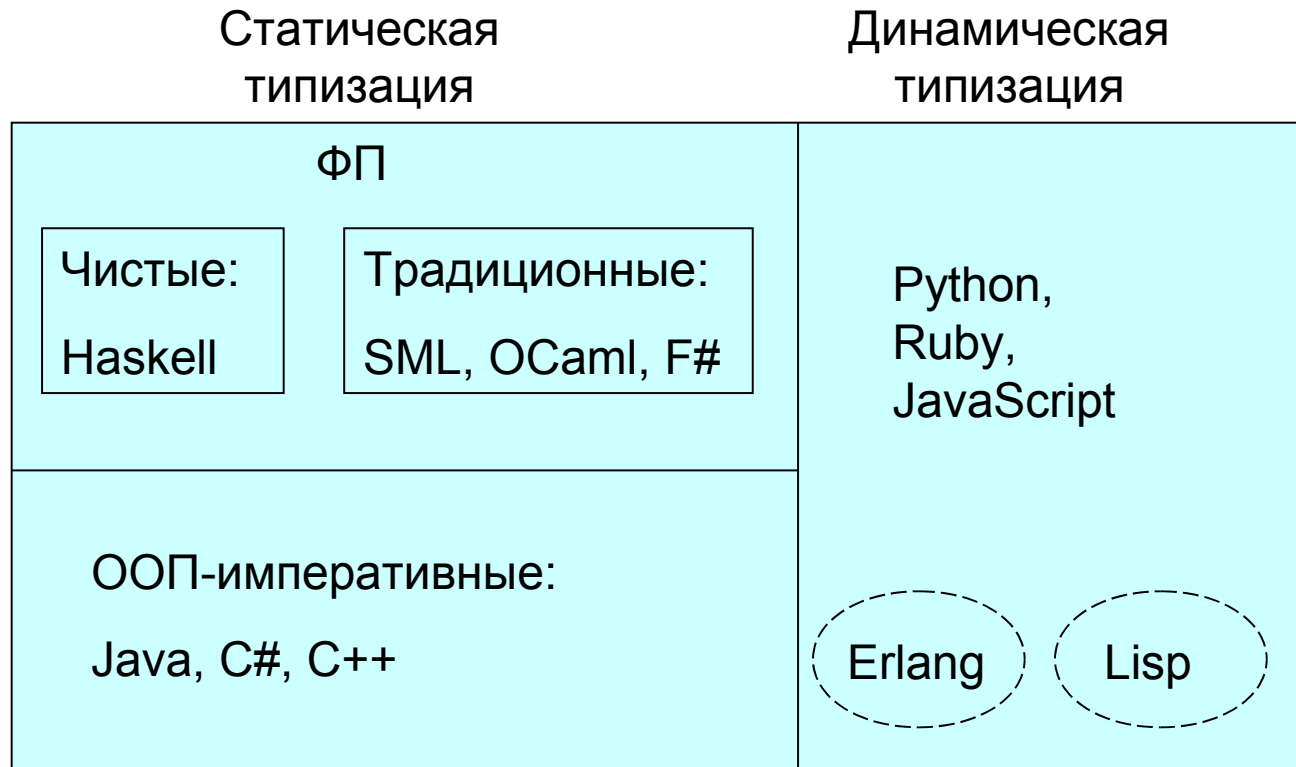
Место Haskell в пространстве дизайна языков

Функции высшего порядка и вывод типов упрощают программирование "в малом", локально. Для того, чтобы понять, чем хорошо программирование на Haskell для проектирования больших программ, нужно сперва рассмотреть, где находится Haskell в пространстве дизайна языков программирования.



Статическая и динамическая типизация

Первое деление, которое здесь видно – это деление на языки со статической/динамической типизацией. Это деление является древнейшей темой религиозных войн в сообществе программистов.



Static vs dynamic typing – с точки зрения фаната динамики



Static = Bondage & Discipline



Dynamic = Гибкость

Static vs dynamic typing – с точки зрения фаната статики



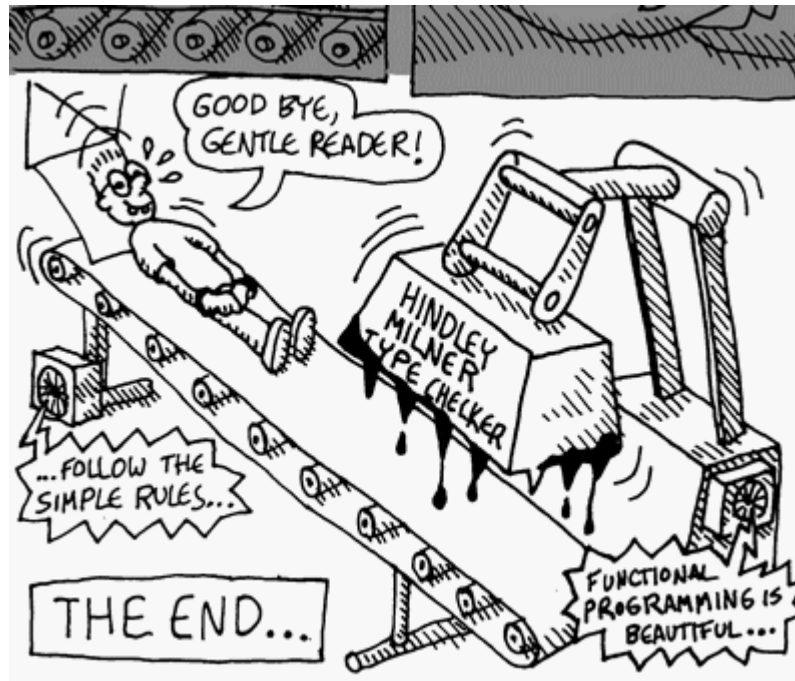
Static = Надежность



Dynamic = Детский сад

Статическая и динамическая типизация

Так как логические аргументы в священных войнах никого не убеждают, то я, как автор, заявляю, что истинно-верной является только статическая типизация, а динамическая ересь подлежит искоренению:



(полный комикс
см. [здесь](#))

Соответственно, убогие недоязыки из правой части даже не подлежат рассмотрению. Некоторого снисхождения заслуживают разве что Erlang и Lisp, так как обладают интересными концепциями вне типизации (метапрограммирование и распределенные вычисления)

Статическая и динамическая типизация

Более серьезно, статическая типизация имеет следующие преимущества:

1. Раннее обнаружение ошибок – на этапе компиляции, а не выполнения. На удивление часто программа на SML или Haskell работает правильно сразу, как только ее наконец удастся скомпилировать.
 2. Поддерживаемость больших проектов при статической типизации намного выше, так как изменения могут быть верифицированы компилятором, и типы являются частью документации программы, облегчающие ее понимание.
 3. Для статически типизированных языков проще делать автоматизированную обработку программ (автоматический рефакторинг, как в средах IDEA или Eclipse).
 4. Для статически типизированных языков проще оптимизация кода, так что в среднем они эффективнее динамически типизированных.
-

FP vs OOP

Практически все популярные императивные языки в настоящее время являются объектно-ориентированными (см. [индекс популярности языков TIOBE](#)). Поэтому в следующих частях будет как сравнение высокоуровневого дизайна программ в ООП и ФП, так и сравнение императивного и функционального программирования в целом.

Но перед тем, как перейти к сравнению, сделаем несколько замечаний по "внутреннему устройству" языков программирования.

Конструкция языков программирования

Одно из свойств языка программирования – ортогональность; его наличие означает, что язык состоит из небольшого числа независимых базовых конструкций, которые можно произвольно комбинировать между собой, получая сколь угодно сложные конструкции.

Примеры не-ортогональности: а) в языке Си нельзя возвращать массив из функции (т.е. конструкции "функция" и "массив" не всегда можно соединить); б) в C++ многие возможности дублируются (функции/методы, указатели/ссылки).

В Haskell его "фичи" на удивление хорошо "пригнаны" друг к другу, там очень мало особых случаев; этот язык обладает ортогональностью. При этом, в отличие от других ортогональных языков (Scheme, Forth), в Haskell есть много вкусного синтаксического сахара, делающего код приятным на вид и хорошо читаемым.

Конструирование языков программирования – непростая задача, и нельзя просто свалить все удачные "фичи" в кучу, чтобы получить "идеальный язык". У Haskell баланс очень удачен, что несомненно скажется на дизайне языков будущего.

Конструкция языков программирования

Краткий список удачных конструкций Haskell и почему их не так просто встроить в другие языки:

- Замыкания (closures) – их трудно полноценно реализовать в языке без сборки мусора типа C++ ([upward funarg problem](#))
- Каррирование (currying) – то же, что для замыканий плюс проблемы с перегрузкой функций по числу аргументов, как принято во многих императивных языках
- Вывод типов (type inference) – глобальный вывод типов накладывает серьезные ограничения на систему типов; так, наличие наследования резко ослабляет возможности вывода типов
- Сопоставление с образцом (pattern matching)
- Классы типов (type classes)

Сравнение императивных и функциональных языков

Проблемы ООП

Прежде всего выявим существующие проблемы в традиционном императивном ООП-подходе, и затем сравним его с подходом, принятым в Haskell.

Сравнение императивных и функциональных языков

Изменяемое состояние (mutable state)

Проблемы изменяемого состояния (mutable state)

В императивном программировании нормой является изменяемое состояние – т. е. если есть объект, то вызовы его методов могут изменять поля метода:

```
class Point {  
    private int x;  
    private int y;  
  
    void move(int dx, int dy) {  
        // изменение состояния объекта:  
        this.x += dx;  
        this.y += dy;  
    }  
}
```

В функциональном программировании изменение состояния является скорее исключением и, как показывает практика, подход ФП позволяет избегать целых классов ошибок программирования.

Проблема нелокальности (aliasing)

Проблема 1. Нелокальность изменений

На один и тот же объект могут быть несколько ссылок (этот эффект называется aliasing). Если объект изменяемый, то изменение по одной ссылке приводит к изменениям по другим ссылкам, но не всегда это то, что нужно.

```
Point p = new Point(1, 2);  
Point p2 = p;  
...  
// Где-то очень далеко - изменение p2:  
p2.move(3, 1);  
...  
// Где-то еще дальше - чтение p:  
int x = p.getX();
```

По тексту программы не видно, что изменение p2 ведет к изменению и p тоже, и это может приводить к неправильной работе программы, особенно если разделяются данные между модулями, написанными разными программистами.

Проблема нелокальности (aliasing)

Так как изменение и использование объекта может происходить в совершенно разных местах программы и в разное время ее выполнения, то отладка подобных ошибок – непростое занятие (один раз пришлось потратить несколько часов на ошибку, связанную с aliasing).

Для борьбы с подобными ошибками используется копирование объектов (defensive copy) – как пример можно посмотреть метод `java.awt.Component.preferredSize()`:

```
public Dimension preferredSize() {  
    Dimension dim = this.prefSize;  
    ...  
    // создание копии:  
    return new Dimension(dim);  
}
```

Но это создает в свою очередь другие проблемы:

- Нужно знать, когда копирование нужно, а когда нет
- Копирование больших объектов может быть неэффективным

Проблема нелокальности (aliasing)

Эта проблема исчезает, если использовать **неизменяемые объекты (immutable objects)** – то есть после создания их состояние не изменяется, а все методы возвращают **новый** объект, а не изменяют существующий.

Плохо

```
class Point {  
    void move(int dx, int dy) {  
        this.x += dx;  
        this.y += dy;  
    }  
}  
  
Point p = new Point(1, 2);  
Point p2 = p;  
p2.move(3, 1);
```

Хорошо

```
class Point {  
    Point move(int dx, int dy) {  
        return new Point(x + dx, y + dy);  
    }  
}  
  
Point p = new Point(1, 2);  
// Только явное изменение –  
// нет "дальнодействия":  
Point p2 = p.move(3, 1);
```

Во втором примере мы явно указываем, что только p2 – измененный объект, и это изменение никак не взаимодействует с остальной программой.

Проблема нелокальности (aliasing)

Если объекты неизменяемые, наличие нескольких ссылок на один объект не создает проблем, такие ссылки можно без опасений передавать и хранить в структурах данных.

К тому же в некоторых случаях это может приводить к более эффективной работе программы:

- Не нужно копирование объектов (defensive copy)
 - Современные сборщики мусора (GC) лучше заточены именно под большое количество объектов с малым сроком жизни, чем под объекты с большим сроком жизни; поэтому можно не бояться создавать новые объекты при вызове методов.
 - Aliasing для изменяемых данных создает проблемы при оптимизации кода, так как запись в память может приводить к изменению закэшированных в регистрах значений (см. [пример](#)). Для неизменяемых данных aliasing не создает проблем.
-

Комбинаторная сложность и поддержание инвариантов

Проблема 2. Комбинаторная сложность и поддержание инвариантов

Как правило, не все состояния объекта являются корректными, а только небольшая их часть. То есть для значений полей объекта должны выполняться условия корректности – инварианты. При наличии методов, которые изменяют состояние объекта, инварианты должны сохраняться.

```
class OrderedPair {  
    private int x, y; // Инвариант:  $x \leq y$   
    OrderedPair(int x, int y) { ... }  
    void setFirst(int newX) { ... }  
    void setSecond(int newY) { ... }  
}
```

В этом примере как конструктор, так и методы setFirst/setSecond должны проверять выполнение инвариантов класса.

Комбинаторная сложность и поддержание инвариантов

Так как в изменяемом объекте любой метод может изменить любое поле, то сложность работы по корректной поддержке инвариантов можно оценить как

$$\text{Число_изменяемых_полей} * \text{Число_методов}$$

При росте числа изменяемых полей проверять корректность изменений становится все сложнее (удержать в голове соотношения даже между тремя полями уже может быть непросто).

Для неизменяемых объектов все резко упрощается – инварианты достаточно проверять только при создании объекта, а любые методы всегда создают новый объект.

Поведение неизменяемых объектов очень простое, так как их состояние не меняется. Понимать программы с их использованием намного проще.

Даже если объект нельзя сделать полностью неизменяемым, ограничение на изменение части его полей упрощает его поведение.

Зависимость от истории

Проблема 3. Зависимость от истории

Корректность состояния изменяемых объектов зависит от порядка вызова методов. Если есть класс с методом инициализации:

```
class MyClass {  
    void init() { ... }  
    void doWork() { ... }  
}
```

```
void f1(MyClass obj) {  
    obj.init();  
    obj.doWork();  
}  
  
void f2(MyClass obj) {  
    obj.doWork();  
}
```

Здесь функцию f1 можно вызвать только для объекта, для которого еще не вызван метод init(), а функцию f2 – только для объекта, у которого уже вызван метод init(). Такие условия невозможно выразить в коде, только в документации (которая обычно устаревает) – это невозможно проверить на этапе компиляции.

Зависимость от истории

Еще хуже, если объект передается между разными модулями, которые ожидают определенной последовательности вызовов метода объекта. Это создает неявные связи между модулями, которые очень трудно тестировать и поддерживать.

Для неизменяемых объектов опять же этой проблемы нет, так как объект никак не меняется при вызове его методов.

Это очень сильно упрощает отладку, так как результат работы метода, работающего только с неизменяемыми полями объекта зависит только от значений этих полей, и не зависит от истории вызовов методов объекта или состояний других объектов.

Хранение объектов в коллекциях

Проблема 4. Хранение объектов в коллекциях

Хранение изменяемых объектов в коллекциях может давать неожиданные результаты:

```
Point p = new Point(1, 2);  
Set<Point> set = new HashSet<Point>();  
set.add(p);  
p.move(-1, 0);  
boolean isInSet = set.contains(p);  
System.out.println(isInSet); // Может быть false
```

Для хранения объекта в хэш-таблице используется значение хэш-функции, вычисленное для начального состояния объекта. После изменения состояния объекта значение хэш-функции для него тоже изменяется, и найти (тот же самый) объект в хэш-таблице уже не получается, так как ищем новый хэш-код, а объект сохранен по старому коду.

Для неизменяемых объектов этой проблемы нет.

Проблема 5. Многопоточный доступ

При доступе к изменяемым полям объекта одновременно из нескольких потоков требуется синхронизация (например, через mutex'ы). Синхронизация потоков – довольно неэффективная операция. Причем чем больше ядер/процессоров в системе, тем большие накладные расходы влечет синхронизация, так как потоки могут выполняться на разных процессорах, а синхронизация требует взаимодействия процессоров.

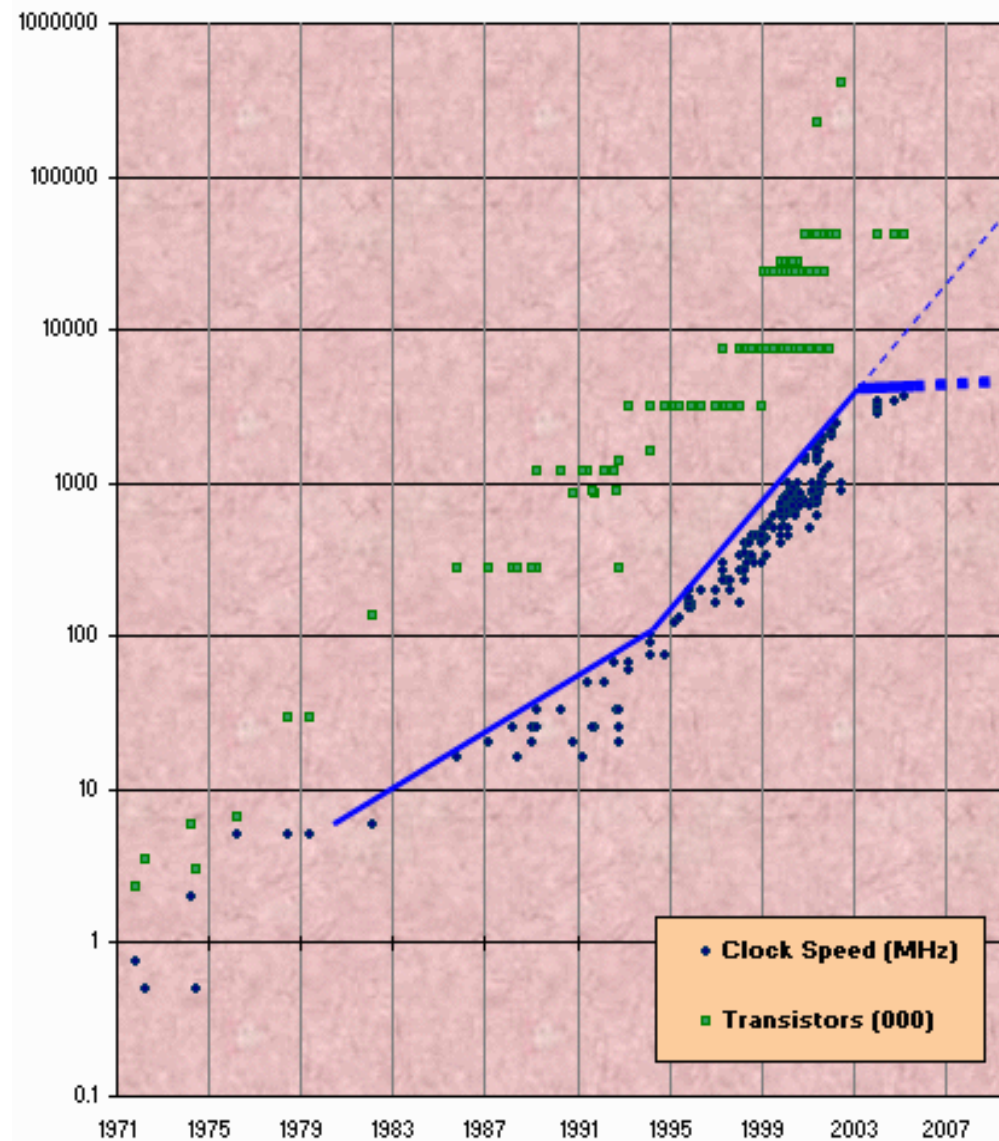
```
class BankAccount {  
    private int amount;  
    synchronized void deposit(int value) {  
        this.amount += value;  
    }  
}
```

К тому же правильная расстановка блокировок, позволяющая избежать deadlock и race condition представляет собой сложную задачу.

Многопоточный доступ

Закон Мура

В последние годы закон Мура ("количество транзисторов на микросхеме увеличивается в 2 раза за 2 года") продолжает действовать, но при этом увеличивается не быстродействие процессора (мегагерцы), а число ядер в процессоре (multicore revolution). Поэтому ускорить выполнение программы можно только распараллеливанием на несколько потоков, выполняющихся на разных ядрах. Из-за этого простота и надежность параллельных вычислений становятся одной из важнейших целей современного проектирования.



Многопоточный доступ

Для неизменяемых данных синхронизация не нужна, так как потоки только читают данные. Использование чисто функциональных структур данных (purely functional data structures) позволяет использовать синхронизацию намного меньше, чем при использовании традиционных изменяемых структур данных.

Так как функциональные языки в основном работают именно с неизменяемыми данными, то они лучше подходят для параллельной обработки данных. Теоретически они также лучше подходят для автоматической параллелизации вычислений:

```
result = map complexFunc list
```

Если функция `complexFunc` не имеет побочных эффектов (а в Haskell это всегда так), то ее можно вычислять для всех элементов списка `list` параллельно. В императивных языках подобная параллелизация сильно затруднена и ограничена небольшим количеством частных случаев.

Но пока автоматическая параллелизация в ФП остается теоретической возможностью, до применения на практике еще далеко.

Многопоточный доступ

Тем не менее параллельные вычисления в функциональных языках проще, так как:

- Отсутствие необходимости расстановки блоков синхронизации значительно упрощает программирование
 - Чистые функции всегда можно выполнять параллельно, так что программисту не надо задумываться над возможными побочными эффектами
 - В Haskell все функции чистые; с помощью монад можно контролировать возможные побочные эффекты
 - В Haskell с помощью монад можно проверять корректность параллелизации кода на этапе компиляции (см. далее про Software Transactional Memory)
-

Immutable data

Хотя достоинства неизменяемых структур данных были известны в среде функционального программирования давно, темные программистские массы начали осознавать их только в конце 1990-х. Например, очень хорошая книга ["Effective Java" Joshua Bloch](#) (2001) включает в себя следующие советы:

- **Favor immutability**
 - Classes should be immutable unless there's a very good reason to make them mutable
 - If a class cannot be made immutable, you should still limit its mutability as much as possible
 - Constructors should create fully initialized objects with all of their invariants established

То есть Блох фактически призывает писать программы на Java в функциональном стиле – неизменяемость данных должна всегда рассматриваться в первую очередь; для облегчения поддержки инвариантов класса все они должны проверяться один раз – в конструкторе класса, после чего объект уже не меняется.

Сравнение императивных и функциональных языков

Объектно-ориентированное
программирование

Наследование и изменяемость

Проблема 1.

Есть известная головоломка в ООП (circle-ellipse problem, или square-rectangle problem):

```
class Rectangle {
    private int w;
    private int h;

    Rectangle(int w, int h) {
        this.w = w; this.h = h;
    }

    void setWidth(int newW) { ... }
    void setHeight(int newH) { ... }
}

class Square extends Rectangle {
    Square(int side) {
        super(side, side);
    }
}
```

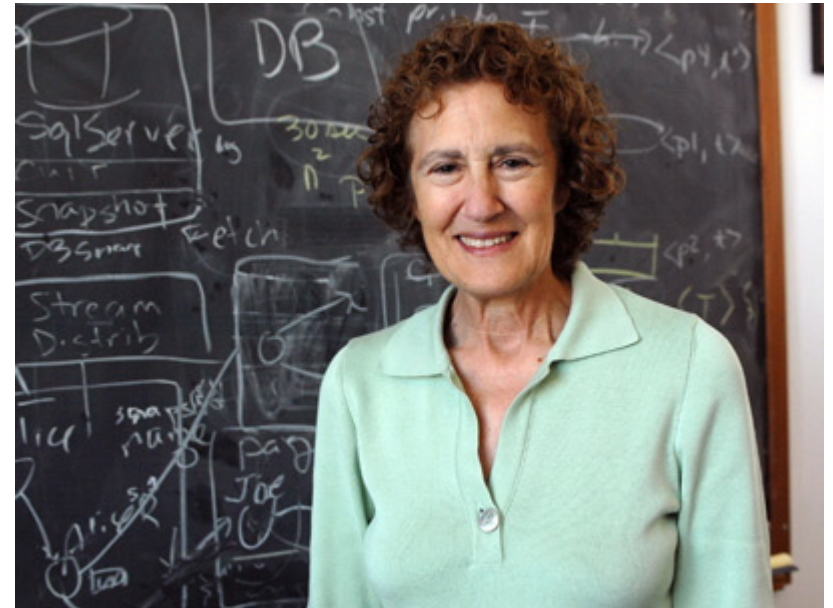
```
void test(Rectangle rect) {
    rect.setWidth(5);
    rect.setHeight(4);
    assert(rect.getWidth() == 5
           && rect.getHeight() == 4);
}

Square square = new Square(3);
test(square); // WTF?!!!
```

Наследование и изменяемость

Так как квадрат является частным случаем прямоугольника, то класс Square должен наследоваться от класса Rectangle, так как отношение наследования моделирует отношение is-a (Square is a Rectangle, но не наоборот).

Но для функции test это не так – она неправильно работает для объектов Square, как бы мы не определили его методы (если не нарушать инвариант $width == height$). Это нарушение Liskov Substitution Principle (LSP): функция, которые принимают значения типа T, должны также обрабатывать и значения любого подтипа T. В нашем случае функция, работающая для прямоугольника, должна работать и для квадрата.



Барбара Лисков

Наследование и изменяемость

Обратное наследование (Rectangle от Square) тоже является ошибочным:

```
class Rectangle extends Square ...  
void test(Square square) {  
    int area = square.getSize() * square.getSize();  
    assert(area == square.getArea());  
}  
Rectangle rect = new Rectangle(5, 3);  
test(rect);
```

Единственное корректное решение проблемы – сделать классы Rectangle и Square неизменяемыми (без методов set...).

Inheritance breaks encapsulation

Проблема 2.

Пример взят из книги Joshua Bloch "Effective Java". Предположим, мы хотим считать, сколько раз в коллекцию добавлялся элемент. Для этого унаследуем класс от класса коллекции:

```
class CountingSet extends HashSet {  
    private int count = 0;  
    public boolean add(Object o) {  
        count++;  
        return super.add(o);  
    }  
    public boolean addAll(Collection c) {  
        count += c.size();  
        return super.addAll(c);  
    }  
}
```

Inheritance breaks encapsulation

Но при использовании `addAll` результат может быть следующим:

```
CountingSet set = new CountingSet();  
set.addAll(Arrays.asList("1", "2", "3"));  
// set.count == 6, хотя добавили 3 элемента!
```

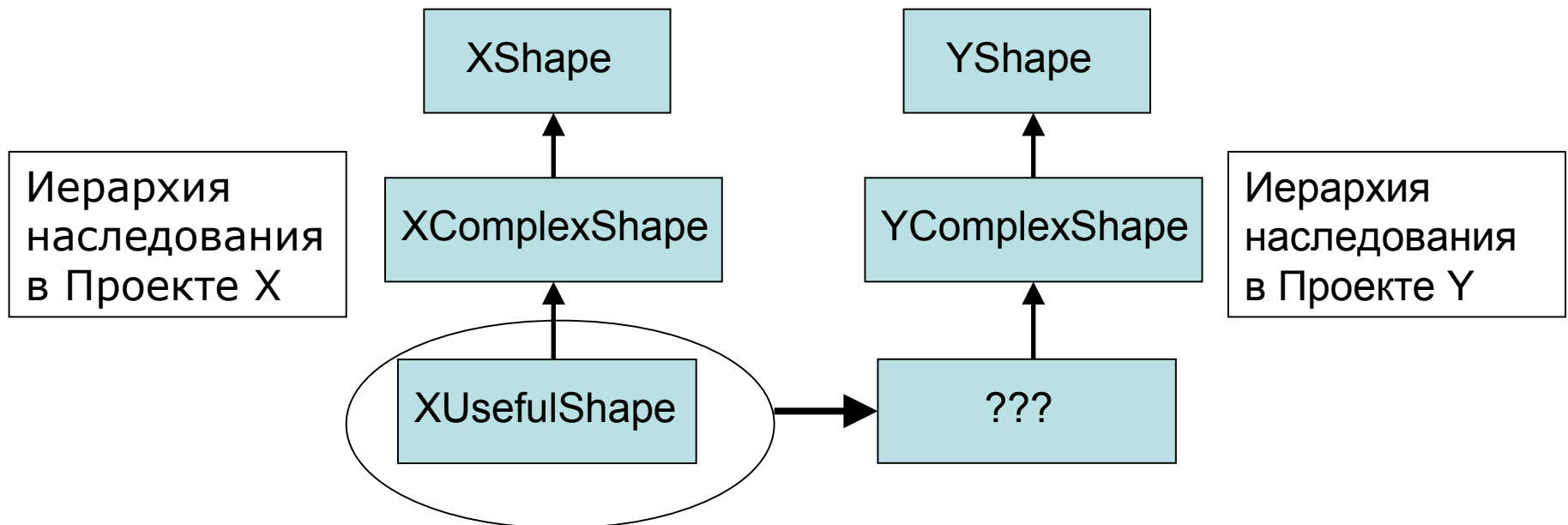
Так как метод `addAll` может быть реализован через последовательный вызов `add`. Если же он реализован по-другому, то результат будет правильным. То есть наследование реализации метода (`override`) приводит к разным результатам в зависимости от реализации базового класса. Таким образом детали реализации прорываются через инкапсуляцию – `inheritance breaks encapsulation`.

Обычное решение этой проблемы – использовать композицию классов вместо наследования (`favor composition over inheritance`).

Проблема 3.

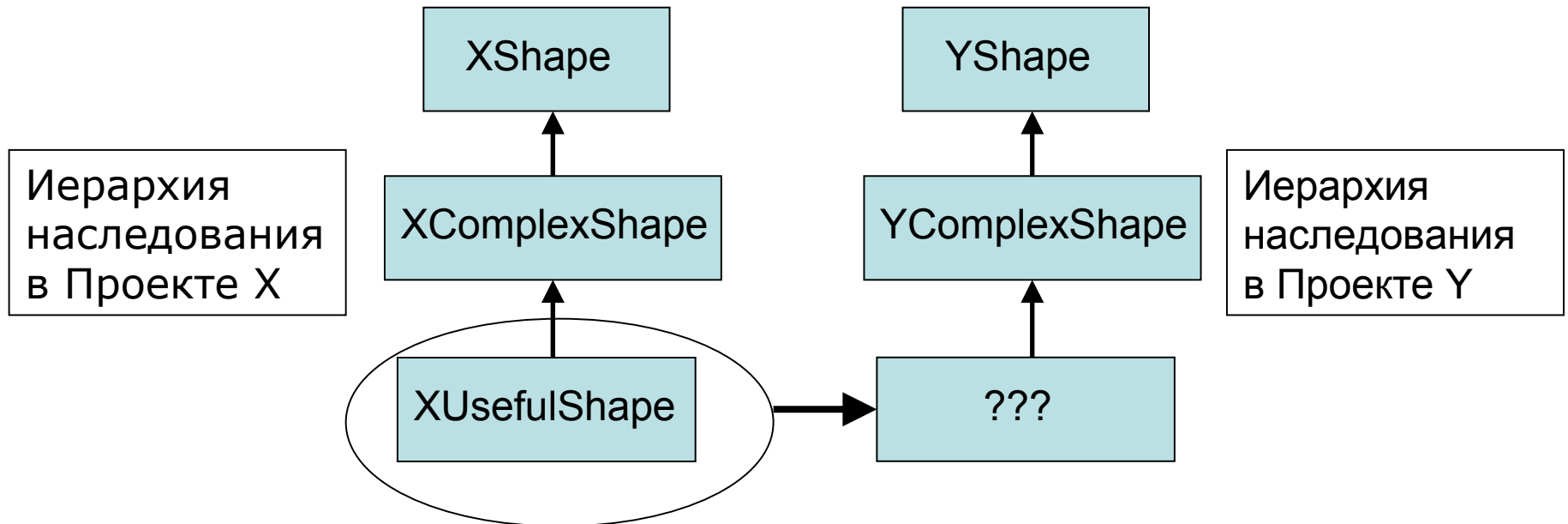
Обычно в обучающих текстах по ООП говорится, что одно из его достоинств – упрощение повторного использования кода (code reuse). На практике чрезмерная приверженность наследованию (а это единственное, что отличает ООП от ФП – инкапсуляция и полиморфизм есть и в Haskell) делает code reuse проблематичным.

Пусть у нас в Проекте X есть класс XUsefulShape, который мы хотим использовать в Проекте Y.



Наследование и code reuse

Но класс XUsefulShape включен в иерархию классов Проекта X, а для того, чтобы взаимодействовать с остальными классами Проекта Y, он должен быть включен в иерархию классов Y. Придется либо перенести целиком всю иерархию X в проект Y (при этом, возможно, часть функциональности будет дублироваться), либо использовать Copy+Paste для создания класса YUsefulShape, заменяя везде X на Y. Любое из этих решений не страдает элегантностью.



Наследование и code reuse

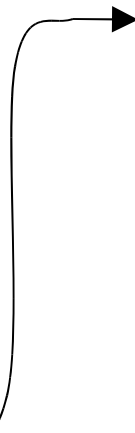
Таким образом, глубокие иерархии классов снижают code reuse, и еще раз подтверждается правило: **Favor composition over inheritance.**

Также, как показывает практика, для композиции классов очень полезным бывает паттерн проектирования "стратегия": пусть мы хотим модифицировать поведение класса А. Вместо того, чтобы оставлять (абстрактные) методы для определения в подклассах, вынесем настройку поведения в отдельный класс/интерфейс и будем хранить его в поле класса А.

Пример: вместо абстрактного класса

```
abstract class Shape {  
    public abstract void paint();  
}
```

МОЖНО СДЕЛАТЬ ТАК:



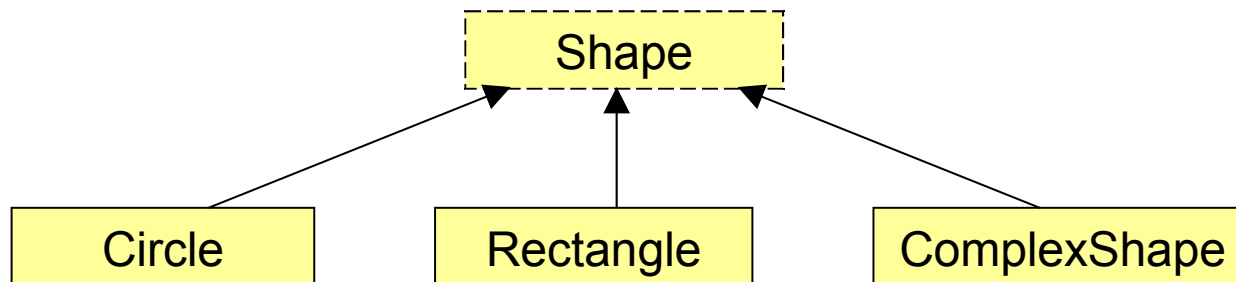
```
interface Painter {  
    void paint(Shape shape);  
}  
  
final class Shape {  
    private final Painter painter;  
    public void paint() {  
        painter.paint(this);  
    }  
}
```

ADT = OOP done right

Итак, для того, чтобы ООП не создавало проблем, структура классов должна удовлетворять следующим свойствам:

- Объекты должны быть неизменяемыми (по возможности)
- Иерархии наследования не должны быть глубокими
- Наследование реализации и переопределение методов использовать не нужно

Визуально подобный дизайн можно представить так:



При этом Shape – это абстрактный класс или интерфейс, методы которого переопределять нельзя, можно только реализовать его абстрактные методы.

ADT = OOP done right

В ООП-коде это будет выглядеть как

```
abstract class Shape ...  
class Circle extends Shape ...  
class Rectangle extends Shape ...  
class ComplexShape extends Shape ...
```

Но то же самое, только короче, можно записать с помощью алгебраических типов данных (algebraic data types, ADT)!

```
data Shape = Circle | Rectangle | ComplexShape
```

При этом все свойства "правильного" ООП будут выполняться автоматически: неизменяемость, отсутствие глубокой иерархии.

ADT = OOP done right

Различия ООП и ADT

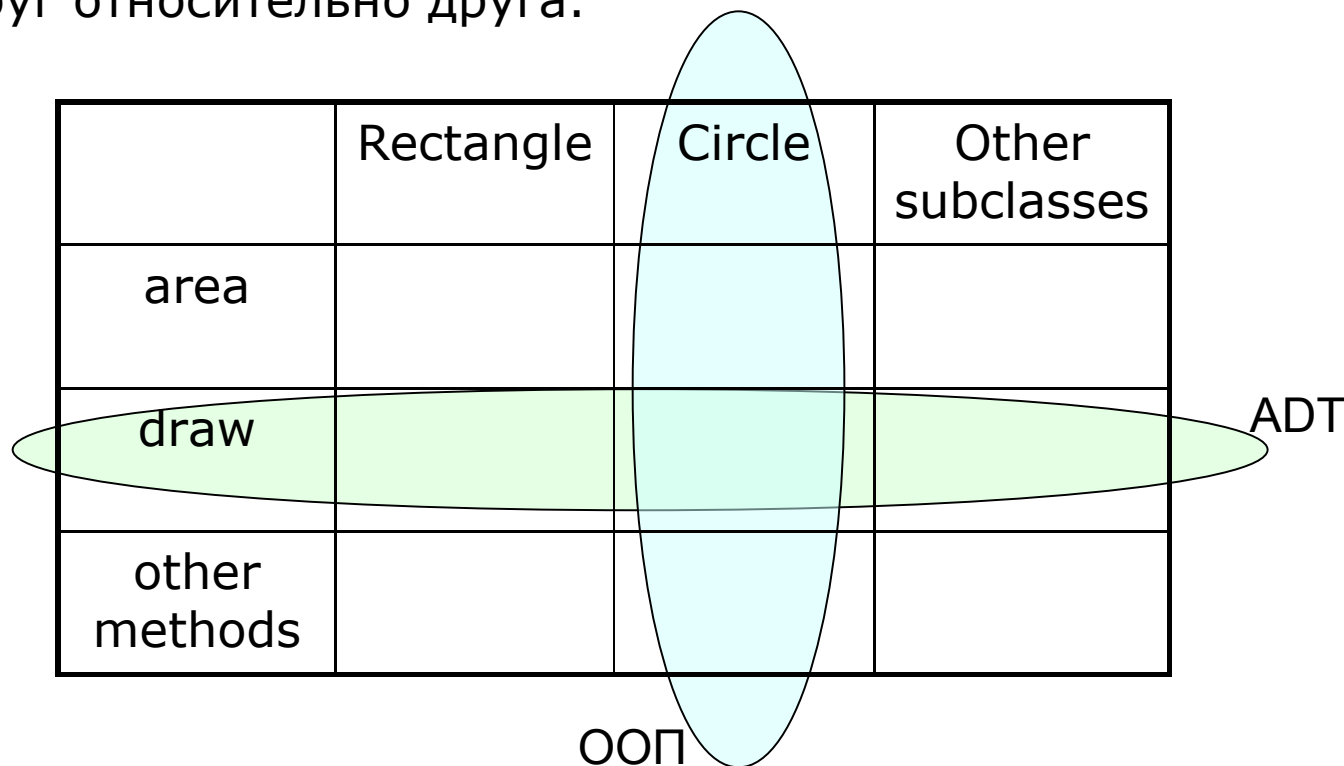
Описание иерархии объектов в ООП и в Haskell отличается:

```
abstract class Shape {  
    abstract float area();  
    abstract void draw();  
}  
  
class Rectangle extends Shape {  
    float w, h;  
    ...  
    float area() { return w * h; }  
}  
  
class Circle extends Shape {  
    float r;  
    ...  
    float area() { return PI * r * r; }  
}
```

```
data Shape =  
    Rectangle Float Float  
    | Circle Float  
  
area :: Shape -> Float  
area (Rectangle w h) = w * h  
area (Circle r) = pi * r * r  
  
draw :: Shape -> IO ()  
...
```

Различия ООП и ADT

В ООП методы группируются по классам; в ADT подклассы группируются по методам (функциям), поэтому они "повернуты на 90°" друг относительно друга.



Из-за этого в ООП легко добавлять новые классы, не меняя код других классов; в ADT – легко добавлять новые функции, не меняя код остальных функций. В разных случаях бывает удобнее либо тот, либо другой подход (ООП или ADT).

ADT = OOP done right

Если мы пишем компилятор, то структура данных, с которой мы работаем – дерево разбора – зафиксирована, а методы обработки могут добавляться неограниченно (новые стадии трансляции, оптимизации, интеграция с IDE). Для этого типа задач больше подходят алгебраические типы данных.

Также ADT удобны для представления классических структур данных – списков, деревьев.

ООП удобно тогда, когда нужна возможность легкого добавления новых сущностей; например, в компьютерных играх – добавление новых персонажей.

То есть оба подхода имеют право на жизнь, но если в ООП довольно легко спроектировать иерархию классов неправильно, то при использовании ADT правильное решение получается как правило "само собой".

Сравнение Haskell и других функциональных ЯЗЫКОВ

Языки семейства ML

Основными конкурентами Haskell среди функциональных языков являются языки семейства ML: [Standard ML](#), [OCaml](#), [F#](#). Их основные отличия от Haskell:

- строгие вычисления вместо ленивых
- возможность написания не-чисто функциональных программ (наличие изменяемых данных и исключений); это позволяет писать программы в императивном стиле
- отсутствие типов классов
- более продвинутая система модулей

Все остальные характеристики функциональных языков (рекурсия, функции высшего порядка, алгебраические типы данных, сопоставление с образцом) у них общие.

Языки семейства ML

Семейство ML-подобных языков создавалось в основном в 80-е – начале 90-х годов. Такая относительная "старость" этих языков имеет и плюсы, и минусы:

- + языки и реализации стабильные и зрелые, годятся для промышленного использования
- ML-подобные языки "вышли из моды", интерес академических исследователей перешел к Haskell; соответственно, сообщество вокруг SML/OCaml значительно уменьшилось
- недостаточно хорошая поддержка многопоточности (многоядерность стала актуальна только в конце 90-х)
- так как исследователи ФП в 80-е годы имели недостаточно опыта, стандартная библиотека SML/OCaml имеет упущения и страдает некоторой непоследовательностью
- отсутствие классов типов приводит к необходимости иметь разные функции для разных типов: вместо одной функции `show` есть функции `Int.toString`, `Float.toString`, `Date.toString` и так далее. Более того, в OCaml для сложения целых чисел – одна операция `(+)`, а для вещественных – другая `(+.)`

Языки семейства ML – синтаксис и семантика

После Haskell синтаксис ML-подобных языков кажется неэлегантным и запутанным:

OCaml

```
let rec sum list =  
  match list with  
  | [] -> 0  
  | x :: xs -> x + sum xs;;
```

Haskell

```
sum [] = 0  
sum (x:xs) = x + sum xs
```

Но более важны различия в семантике:

- ML: строгий, нечистый
- Haskell: ленивый, чистый

Далее рассмотрим эти противопоставления отдельно (чистый-нечистый, строгий-ленивый), так как они независимы друг от друга (могут существовать одновременно строгие и чистые языки, хотя пока существуют только экспериментальные версии таких языков)

Функциональная чистота

Функция является чистой, если ее значение зависит только от значений входных параметров. Чисто функциональный язык – язык, в котором можно использовать только чистые функции.

Наличие в языке переменных противоречит чистоте:

```
int variable = 0;
// функция f - нечистая
int f(int x) {
    variable++;
    return x + variable;
}
print(f(0)); // 1
print(f(0)); // 2 - отличается от предыдущего значения
```

В ML-подобных языках переменные есть:

```
val variable = ref 0
fun f x = (variable := !variable + 1; x + !variable)
```

В Haskell переменных нет, хотя их можно эмулировать с помощью монад.

Функциональная чистота

То есть чистота является ограничением для языка – на нечистом языке всегда можно писать и чистые программы. Что же дает функциональная чистота?

Важным для функциональных программ является свойство referential transparency (прозрачность по ссылкам). Его наличие означает, что вместо ссылки на функцию можно подставить ее определение:

```
twice x = x + x  
test = twice 10  
  
-- можно преобразовать в:  
test = 10 + 10
```

Для чистых функций это свойство всегда выполняется. При наличии нечистых функций это не так:

```
twice x = x + x  
test = twice readInt  
  
-- readInt - не чистая функция, так что приведенное не эквивалентно  
-- следующему (два чтения числа вместо одного):  
test = readInt + readInt
```

Функциональная чистота

Referential transparency дает большую свободу для компилятора по преобразованию программы – используя подстановку тела функции вместо ссылки на нее, можно гарантировать эквивалентность следующих преобразований:

```
map f (map g list) = map (f . g) list  
filter f (filter g list) = filter (\x -> f x && g x) list
```

Такие преобразования позволяют уменьшить количество промежуточных объектов при работе со списками и другими структурами данных – так, в первом примере вместо двойного применения функции `map` к списку она применяется только один раз, т.е. промежуточный список не создается.

Такие оптимизации (как и их более общие варианты `deforestation` и `stream fusion`) доступны только в чистых языках.

Функциональная чистота – плюсы

Итак, что нам дает функциональная чистота:

- БОльшие возможности для оптимизации
 - чисто функциональные программы легче понимать и отлаживать, так как результат функции зависит только от входных значений – отлаживать функцию можно независимо от остальной программы, при этом ее всегда можно разбить на меньшие подфункции и отладить их отдельно
 - чистые функции намного проще объединять в композиции, так как между ними нет неявных связей в виде изменяемого состояния (см. классическую статью "[Why Functional Programming Matters](#)")
 - как уже было сказано ранее, чисто функциональные программы проще сделать параллельными; во всяком случае, чистая функция всегда является thread-safe, т.е. ее можно вызывать из нескольких потоков одновременно
-

Функциональная чистота – минусы

Минусы функциональной чистоты:

- трудности при взаимодействии с внешним миром (ввод/вывод – I/O). В Haskell решается через монады – работающий, хотя и довольно громоздкий способ
 - но монады в силу своей "вирусной" природы начинают загромождать всю программу
 - в случае повсеместного использования монады IO мы фактически делаем программу императивной; при этом многие функциональные оптимизации уже применить нельзя
- некоторые алгоритмы и структуры данных (массивы, хэш-таблицы) не имеют чисто-функциональных аналогов по эффективности – так, функциональный аналог массива имеет скорость доступа $O(\log N)$ вместо $O(1)$

Функциональная чистота

Из-за нарастающей важности параллельных вычислений баланс плюсов и минусов начинает складываться в пользу чистых языков. Так, Тим Суини предлагает использовать для физических расчетов в игровых движках именно чисто функциональный подход. Почему? Потому что в нечистом языке параллелизм создает большие проблемы:

```
int globalState = 0;

int f(int x) {
    // неизвестно, меняет ли эта функция глобальное состояние или нет
}

void runningInThread() {
    // можно ли безопасно вызывать эту функцию?

    int t = f(10);
}
```

Нечистые функции в существующих языках внешне никак не отличаются от чистых, так что очень трудно понять, безопасно ли вызывать их из разных потоков, и компилятор не может в этом помочь. Так, [проект STM.NET провалился](#) именно по этой причине. В Haskell же решение – монады – есть, хотя и не очень удобное.

Функциональная чистота

Одно из возможных решений в будущих языках – более удобная, чем монады, система деления функций на чистые и нечистые (**контроль побочных эффектов**). Некоторое подобие такой системы есть в языке **D**. Более общий подход – система эффектов – реализован в экспериментальном языке **Disciple**.

Во всяком случае, если ранее по умолчанию любая функция была нечистой, и для ограничения ее эффектов использовались специальные модификаторы типа `const`, то в будущем скорее всего чистота и неизменяемость будут приняты по умолчанию, а для возврата к старому доброму императивному стилю это придется явно указывать.

Haskell в данном отношении ближе к идеалу, чем ML-подобные языки.

Ленивость

Строгие вычисления: перед вызовом функции значения ее аргументов вычисляются (call-by-value):

```
f x y = y  
f (1+2) (3+4)  
-> f 3 7  
-> 7
```

Ленивые вычисления: значения аргументов не вычисляются, аргументы передаются в виде невычисленных thunk'ов, которые форсируются по мере необходимости (call-by-need):

```
f x y = y  
f (1+2) (3+4)  
-> (3+4)  
-> 7
```

В последнем примере выражение $(1+2)$ не вычисляется, так как его значение не используется.

Ленивость

Ленивый язык обязан быть чистым, но обратное неверно – может быть строгий чистый язык.

Какие преимущества у ленивости для чисто функционального языка? Ленивый язык будет более выразительным, чем строгий. То есть любая чисто функциональная программа со строгими вычислениями будет работать для ленивых вычислений, но не наоборот. Элементарный пример:

```
f x y = y  
-- работает только в ленивом языке:  
f (1/0) 10
```

Что ленивый язык позволяет делать:

- можно присваивать значения в любом порядке, т.к. при ленивых вычислениях вычисление значения произойдет только при его использовании
- можно использовать бесконечные структуры данных (бесконечные списки, бесконечные деревья); можно манипулировать ими так же, как и обычными – главное, чтобы никогда не вычислялся весь список целиком

Ленивость

Также иногда можно улучшить эффективность использования памяти при ленивых вычислениях:

```
sum [1..1000000]
```

При суммировании ленивого списка его элементы вычисляются по мере надобности, и хотя в конце концов будут вычислены все элементы `1..1000000`, они будут вычислены по очереди, и использование памяти будет $O(1)$, в отличие от $O(N)$ в строгой версии. То есть использование памяти "размазывается" по времени.

Но это свойство может иметь и неожиданные последствия:

```
-- также вычисляется в  $O(1)$  памяти:  
test = sum [1..1000000] / fromIntegral (length [1..1000000])  
  
-- А это уже нет - использует  $O(N)$  памяти:  
test = sum list / fromIntegral (length list)  
    where list = [1..1000000]
```

Во втором случае список `list` используется два раза, так что он целиком накапливается в памяти для второго использования.

Ленивость

Таким образом, весьма невинно выглядящие преобразования программы на ленивом языке могут приводить к радикальным изменениям использования памяти. Потребление памяти программой на Haskell может быть очень трудно предсказуемым.

Для борьбы с этими эффектами в Haskell используются различные аннотации строгости (и функции `$!` и `seq`), то есть Haskell – язык ленивый по умолчанию с опциональной строгостью.

В ML-подобных языках наоборот – строгость по умолчанию с опциональной ленивостью (впрочем, то же можно сказать обо всех остальных строгих языках типа Java или C++ – там элементарная ленивость достигается с помощью `if/then/else`).

Интересной промежуточной стратегией вычислений являются "расслабленные вычисления" ([lenient evaluation](#)), при которой аргументы вычисляются не до вызова функции и не по требованию, а параллельно вычислению функции.

Строгость

В целом, похоже, баланс мнений в ФП-сообществе складывается не в пользу ленивости по умолчанию – большинство считает, что повышение выразительности языка недостаточно велико по сравнению с проблемами поиска утечек памяти, тем более что с опциональной ленивостью можно получить большинство преимуществ ленивости без ее проблем.

Некоторые же (как Тим Суини) считают, что будущее за *lenient evaluation* – эта стратегия лучше всего подходит для параллельных вычислений.

Так что с точки зрения практического применения ленивость Haskell представляет скорее проблему; но как языку для изучения это скорее придает ему больший интерес.

Идеальный функциональный язык

Итак, в гипотетическом идеальном функциональном языке должно быть:

- чистота по умолчанию, с возможностью явно указывать возможные побочные эффекты функций (Haskell близок к этому)
- строгость по умолчанию с удобной опциональной ленивостью (здесь Haskell промахнулся, но этот опыт тоже ценен для ФП-сообщества)

Уроки Haskell

Уроки Haskell

Для того, чтобы применять принципы функционального программирования, не обязательно использовать именно Haskell – эти принципы можно использовать в любом языке. Это может быть Scala, F#, Clojure, C# или даже Java.

Изучение функционального программирования позволяет взглянуть на проектирование программ с другой стороны, что расширяет кругозор и набор применяемых "инструментов". Haskell, как наиболее чистое воплощение идей ФП, лучше всего подходит для этой цели.

То есть цель изучения Haskell – не только и не столько сам Haskell, а понимание более общих концепций, которые универсально применимы в программировании.

Immutability – еще раз!

В очередной раз повторяю – неизменяемые объекты реально помогают в программировании – их проще проектировать, отлаживать, изменять.

То есть когда вы создаете новый класс в программе, по умолчанию он должен быть неизменяемым (для Java это означает `final` для всех полей класса). Только при большой необходимости стоит разрешать изменяемые поля.

В принципе, чтобы понять ценность неизменяемости, даже не нужен Haskell – лично мой стиль программирования сдвинулся в сторону неизменяемости еще до Haskell; аналогичный сдвиг произошел в головах многих разработчиков в конце 1990-х – начале 2000-х – достаточно сравнить оригинальный Java API с ужасными классами типа `java.util.Date`, спроектированными в середине 1990-х, и современные библиотеки.

Функции высшего порядка

ФВП – один из основных инструментов функционального программирования, и он действительно упрощает разработку "в малом" – сокращает запись циклов и позволяет проще комбинировать функции. Во многих современных языках, к счастью, есть поддержка удобной записи лямбда-выражений, что позволяет довольно удобно использовать ФВП.

К сожалению, в Java приходится пользоваться очень громоздкой записью; Python ограничивает запись лямбда-выражений одной строчкой.

Так что по возможности нужно использовать ФВП – например, для обработки коллекций, но без фанатизма – если запись в вашем языке (как в случае Java) становится слишком громоздкой, лучше написать банальный императивный цикл.

Влияние ФП-стиля на проектирование ООП уже было обсуждено: нужно придерживаться неглубоких иерархий наследования, по возможности использовать композицию классов вместо наследования. Для вынесения меняющегося поведения из иерархии классов используется паттерн "Стратегия" из [Gang of Four](#).

Кстати, можно заметить, что по сути паттерн "Стратегия" является формой использования функций высшего порядка в ОО-программах – только здесь мы передаем поведение не в виде функции, а в виде интерфейса.

Ленивость

Для обработки коллекций может быть удобно использовать ленивые коллекции для того, чтобы избежать создания промежуточных объектов. Очень удачно это сделано в LINQ в .NET.

Можно отметить, что итераторы в C#, генераторы в Python фактически представляют собой ленивые последовательности (streams).

Что изучать дальше

QuickCheck

Мы изучили только стандарт языка и библиотеку Prelude, возможности же GHC и его стандартных/нестандартных библиотек намного шире.

Пример – библиотека QuickCheck для быстрого тестирования (входит в Haskell Platform).

Использование QuickCheck:

```
Файл test.hs:
take' 0 _ = []
take' _ [] = []
take' n (x:xs) = x : take' (n - 1) xs

ghci test.hs
GHCi, version 6.10.4: http://www.haskell.org/ghc/ :? for help
*Main> import Test.QuickCheck
*Main Test.QuickCheck> quickCheck (\n list -> take' n list == take n list)
Falsifiable, after 0 tests:
-1
[()]
```

Здесь мы убеждаемся, что наше определение take' отличается от стандартного take для параметров -1, [()].

Стандартная библиотека

В стандартной библиотеке GHC реализованы некоторые часто используемые структуры данных – списки, отображения (maps), множества, деревья. Большая часть из них является чисто функциональными (purely functional data structures; или, как в последнее время их модно называть – persistent data structures; persistent в данном контексте означает, что изменение структуры данных порождает новую версию данных, оставляя старую неизменной). Как показали [исследования](#), такие структуры данных могут быть реализованы весьма эффективно. Примеры:

- Пакет Data.List – функции для списков (некоторые из них доступны и в Prelude)
 - Пакет Data.Map – эффективная реализация отображений (maps)
 - Пакет Data.Set – реализация множеств
 - Пакет Data.Tree – функции работы с деревьями
 - Пакеты Data.IntMap, Data.IntSet – еще более оптимизированные варианты map и set для целых чисел
-

Стандартная библиотека

Кроме функциональных структур данных, ради большей эффективности можно использовать и традиционные императивные изменяемые структуры данных. Для этого, естественно, приходится использовать монаду IO.

- Пакет `Data.IORef` позволяет создавать переменные, т.е. ячейки с содержимым, которое можно менять
- Пакет `Data.Array` – доступ к классическим императивным массивам
- Пакет `Data.HashTable` – хэш-таблицы

Реализация строк в виде списка символов (`type String=[Char]`) красива, но, увы, неэффективна. Поэтому в большинстве реальных программ используется пакет `Data.ByteString`, предоставляющий эффективную реализацию строк.

Для прагматического программирования совершенно необходим пакет файлового ввода-вывода IO.

Стандартная библиотека

Также полезными могут оказаться пакеты `Data.Maybe` и `Data.Either`, часто используемые для вычислений с возможными ошибками.

Кроме класса типов `Monad` в стандартной библиотеке присутствуют и другие классы-абстракции вычислений:

- `Functor` и `Applicative` - более "слабые", но и более общие классы, чем `Monad`; для многих операций с монадами на самом достаточно более слабого класса `Applicative`
- `Foldable` и `Traversable` – для обобщения обработки любых структур данных
- `Monoid` – абстракция, соответствующая моноиду в теории групп
- `MonadPlus`, `Arrow`, `Category`

Многие из этих классов имеют происхождение из теории категорий; их описание и ссылки на материалы для дальнейшего изучения можно найти в [Typeclassopedia](#).

Список стандартных монад тоже можно расширить: это не только `Maybe`, `State`, `IO`, список, но и `Reader`, `Writer`, `Cont`.

Инструменты для Haskell

Кроме компилятора и интерпретатора для Haskell есть и другие инструменты:

- [Hoogle](#) – поиск функций по их типу
- [HLint](#) – поиск частых ошибок в программах на Haskell
- [Cabal](#) – система установки внешних пакетов и [Hackage](#) – репозиторий пакетов; так, Hoogle и HLint можно установить с помощью Cabal:

```
$ cabal update  
$ cabal install hlint
```

Продвинутые концепции

Кроме монад, в Haskell в более узких областях есть другие интересные концепции:

- Парсеры на комбинаторах (parser combinators) – то, что в других языках делается с помощью Lex/YACC/ANTLR, в Haskell можно сделать в рамках языка – см. пакеты Text.ParserCombinators и статью "[Parsing with Haskell](#)" для введения в курс дела
- Monad transformers – используются для композиции монад, когда побочный эффект функции складывается из нескольких эффектов
- Zipper – "указатель" в структуре данных; например, если хранить позицию в списке в виде индекса, то каждый доступ будет $O(N)$; если же хранить ее в zipper'e, то $O(1)$
- Iteratee I/O – используется тогда, когда нужна высокая эффективность ввода-вывода; также один из примеров, когда потребность в монадах возникает естественным образом; см. [презентацию](#)
- Извращения с типами – [фантомные типы и обобщенные ADT \(GADTs\)](#), [экзистенциальные типы](#), семейства типов и т.д.

Светлое будущее

Светлое будущее

Академическое сообщество вокруг ФП продолжает исследования, результаты которых когда-нибудь станут такими же привычными, как сборка мусора сегодня. Некоторые примеры:

- **Software Transactional Memory (STM)** – позволяет обходиться без явных блокировок при многопоточном программировании (в Haskell – пакет `Control.Concurrent.STM`; также реализована в **Clojure**)
- Зависимые типы (**dependent types**) – позволяют обнаруживать ошибки выхода за границы массива на этапе компиляции; язык с поддержкой **dependent types** – **ATS**
- Гибридизация ФП и ООП – язык **Scala** объединяет в себе обычные классы, алгебраические типы данных (в виде `case classes`), `traits`
- Язык **Disciple** вместо монад вводит **систему эффектов**, которая позволяет более просто выражать побочные эффекты функций, не нарушая простоты их композиции

Светлое будущее

Проект [Reduceron](#) направлен на создание специализированного процессора для выполнения Haskell-программ; средние результаты выполнения тестовых программ для текущего прототипа: GHC -O2 на процессоре Intel 3000 MHz работает в 3 раза быстрее, чем Reduceron на частоте 96 MHz. Если считать, что скорость будет пропорциональна частоте, то Reduceron на частоте 3000 MHz будет в 10 раз быстрее GHC с максимальной оптимизацией, что также будет быстрее многих программ на Си.

Ссылки

Ссылки

Языки:

- ATS: <http://ats-lang.sourceforge.net>
 - Clojure: <http://clojure.org>
 - D: <http://www.digitalmars.com/d>
 - Disciple: <http://www.haskell.org/haskellwiki/DDC>
 - F#: <http://research.microsoft.com/en-us/um/cambridge/projects/fsharp>
 - Standard ML of New Jersey: <http://www.smlnj.org>
 - OCaml: <http://caml.inria.fr>
 - Scala: <http://www.scala-lang.org>
-

Ссылки

Статьи и презентации (приблизительно в порядке возрастания сложности):

- Tim Sweeney, "[The Next Mainstream Programming Language](#)"
- Simon Peyton Jones, "[A Taste of Haskell](#)"
- Евгений Кирпичёв, "[Элементы функциональных языков](#)"
- John Hughes, "[Why Functional Programming Matters](#)"
- Lennart Andersson, "[Parsing with Haskell](#)"
- Simon Peyton Jones, "[Beautiful Concurrency](#)"
- Simon Peyton Jones, "[Wearing the hair shirt: a retrospective on Haskell](#)"
- Simon Peyton Jones, "[Caging the effects monster: the next big challenge](#)"
- Ben Lippmeier, "[Type Inference and Optimisation for an Impure World](#)"
- Brent Yorgey, "[The Typeclassopedia](#)"
- Oleg Kiselyov, "[Iteratee IO – safe, practical, declarative input processing](#)"