Haskell — общая точка зрения

История

История языков функционального программирования начинается с исследований Алонсо Чёрча, Хаскелла Карри, Стивена Клини, где-то с конца 30-х гг. XX века. Комбинаторная логика, лямбда-исчисление, частично-рекурсивные функции — вот примерно каким было то теоретическое основание.

Но в конце 50-х гг. Джон МакКарти разработал язык Lisp (1958), который фактически и стал первым языком функционального программирования.

Рис. 1: (это не Джон МакКарти)

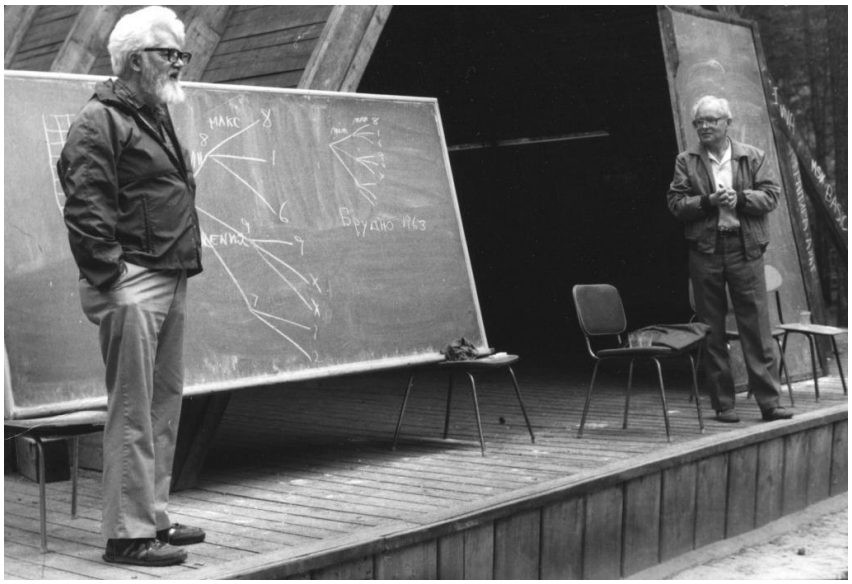

Рис. 2: (Джон МакКарти в Академе)

Потом их стало очень много: APL (1964), ISWIM (1966), FP (1977), ML (1973) — первый язык с типизацией Хиндли–Милнера, Scheme (1975) — это один из двух самых популярных диалектов языка Lisp, Miranda (1985) — один из первых ленивых языков, Hope (1980) — один из первых языков с алгебраическими типами данных.

История языка Haskell начинается в 1987 году. Один за другим появлялись новые функциональные языки программирования. После выхода Miranda (Research Software Ltd, 1985 год) возрос интерес к ленивым вычислениям: к 1987 году возникло более дюжины нестрогих чисто функциональных языков программирования.

Miranda использовался наиболее широко, но это было запантетованное ПО. На конференции по функциональным языкам программирования и компьютерной архитектуре (FPCA, 1987) в Портленде (Орегон) участники пришли к соглашению, что должен быть создан комитет для определения открытого стандарта для подобных языков. Целью комитета являлось объединение существующих функциональных языков в один общий, который бы предоставлял базис для будущих исследований в разработке функциональных языков программирования.

Так появился Haskell. Он был назван в честь одного из основателей комбинаторной логики Хаскела Карри (Haskell Curry).

Рис. 3: Haskell Curry

К концу 1980-х годов было создано много функциональных языков. Часть из них оказали значительное влияние на Haskell:

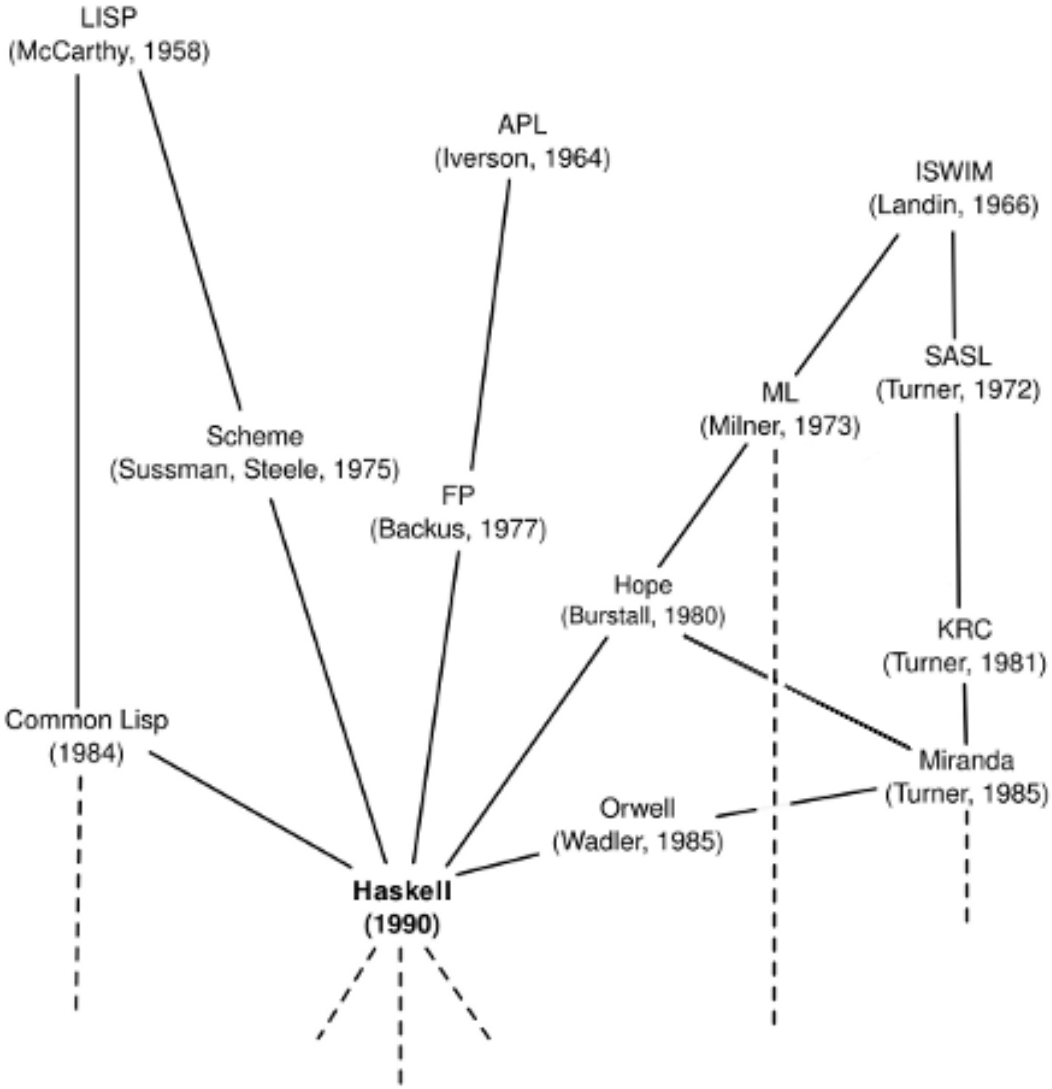

Рис. 4: Haskell History

habr: История языков программирования: как Haskell стал стандартом функционального программирования

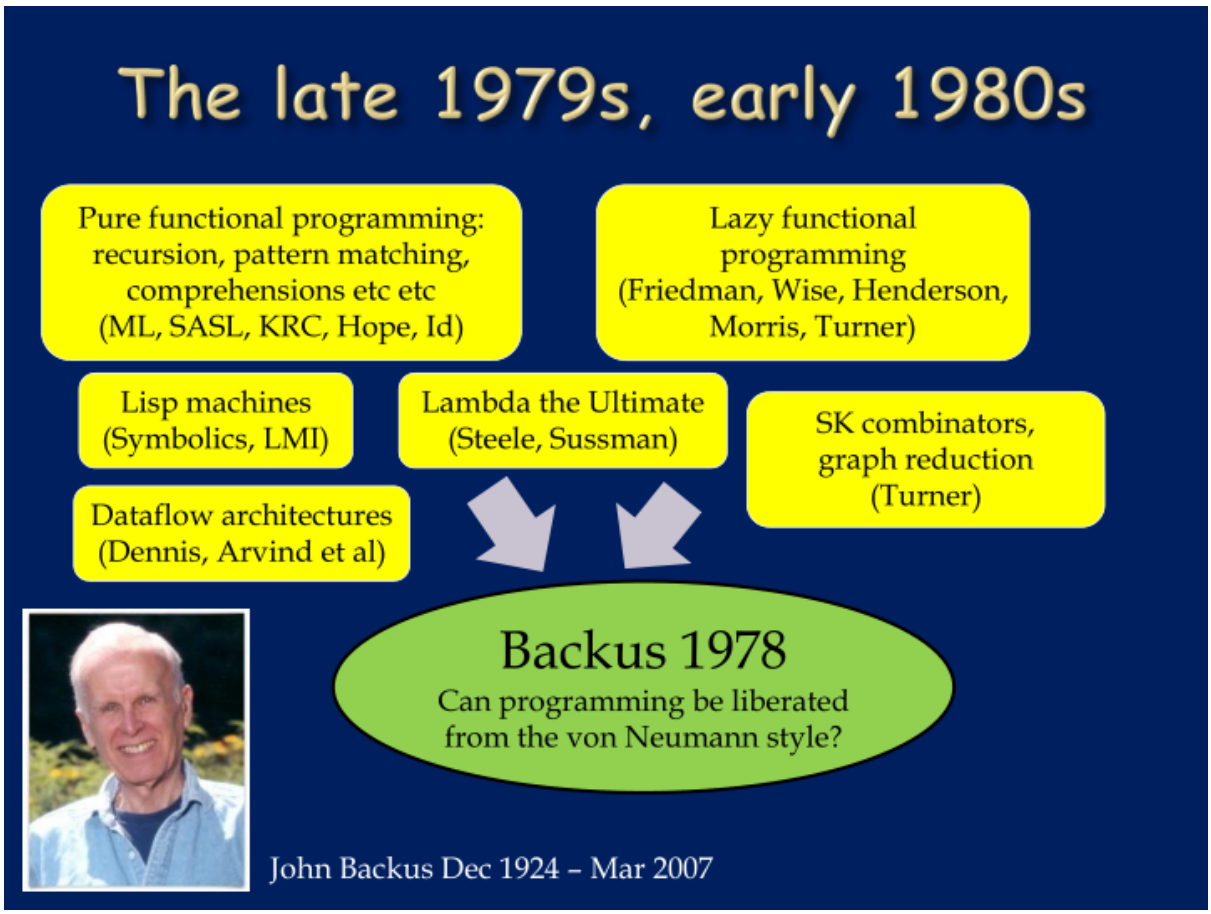

Рис. 5: тезис Бэкуса

История развития языка сама по себе тоже необычна:

Например, в результате развития «пакетной дистрибьюции»:

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/ECOOP-July09.pdf

И да, спонсор: Microsoft Research Lab:

https://www.microsoft.com/en-us/research/lab/microsoft-research-cambridge/

Особенно в части развития компилятора GHC (Glasgow Haskell Compiler)

https://www.haskell.org/ghc/

Особенности языка

ru.wiki: Haskell

В качестве основных характеристик языка Haskell можно выделить следующие:

- использование функций как основных программных сущностей, функции высших порядков (возможность передавать функции в качестве аргументов другим функциям) и лямбда-абстракции;

Рис. 6: процесс развития большинства новых языков

Рис. 7: обычный процесс развития исследовательских языков

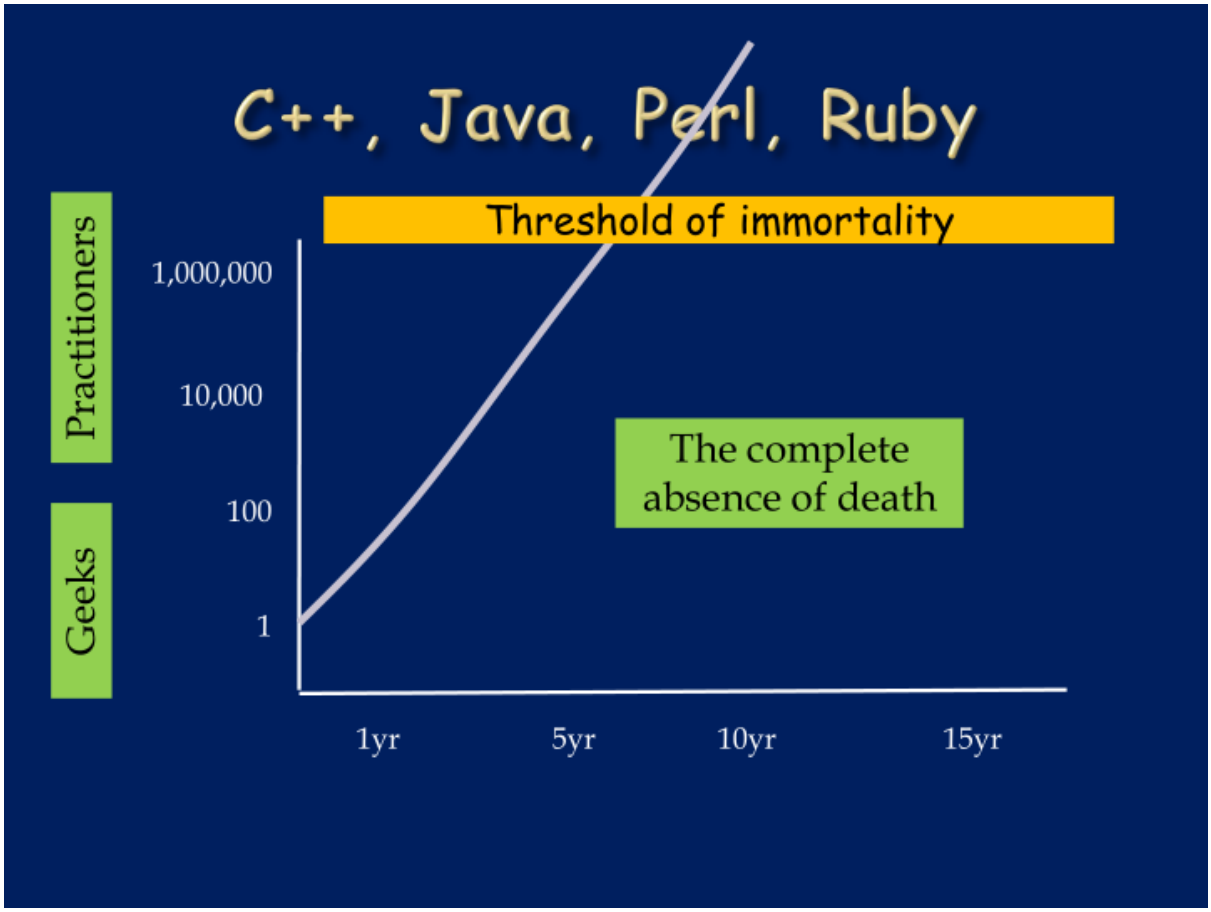

Рис. 8: бессмертные языки

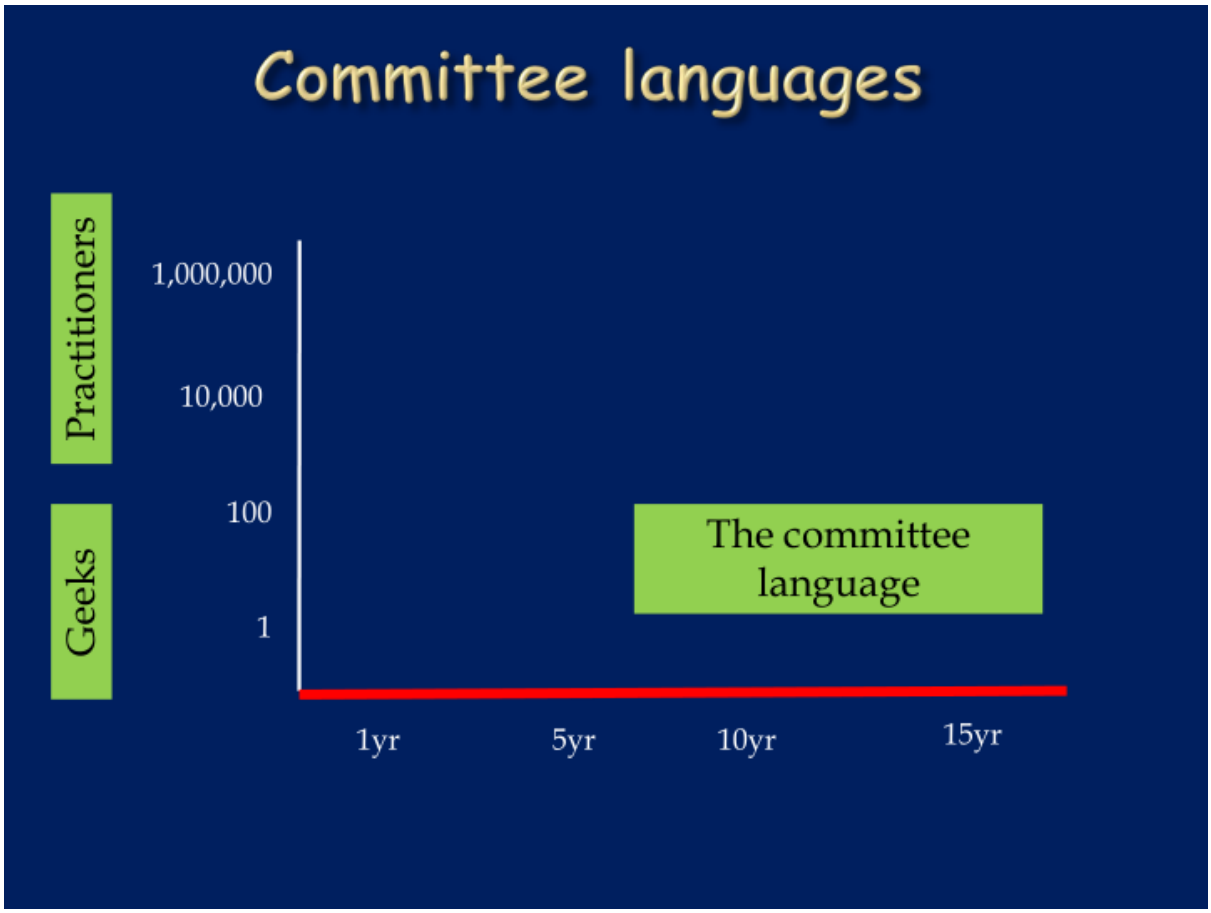

Рис. 9: языки, созданные комитетами

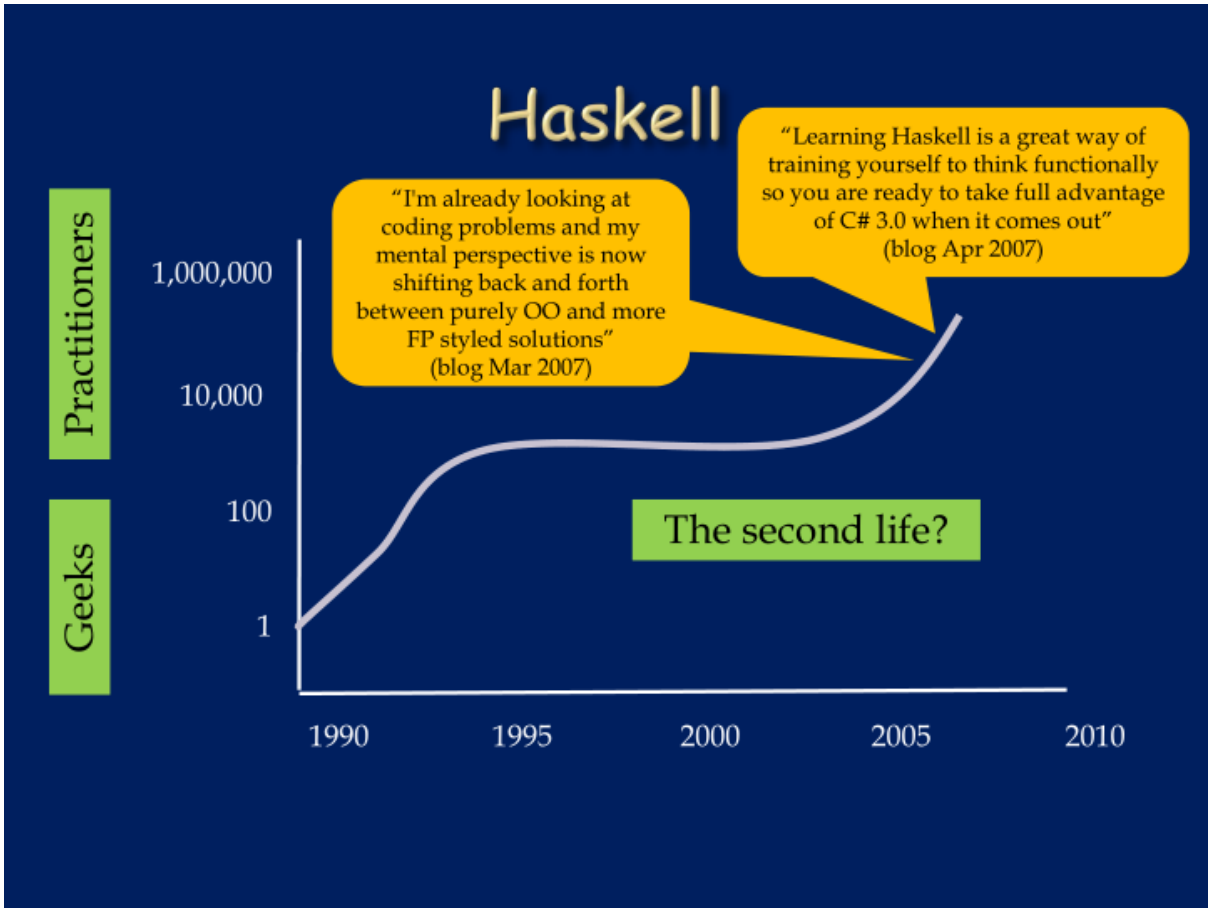

Рис. 10: вторая жизнь Haskell

Mobilising the community

- Package = unit of distribution
- **Cabal**: simple tool to install package and all its dependencies

```
bash$ cabal install pressburger
```

- **Hackage**: central repository of packages, with open upload policy

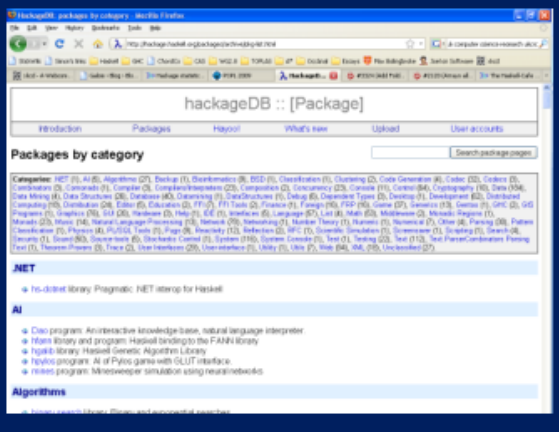

Рис. 11: вторая жизнь Haskell

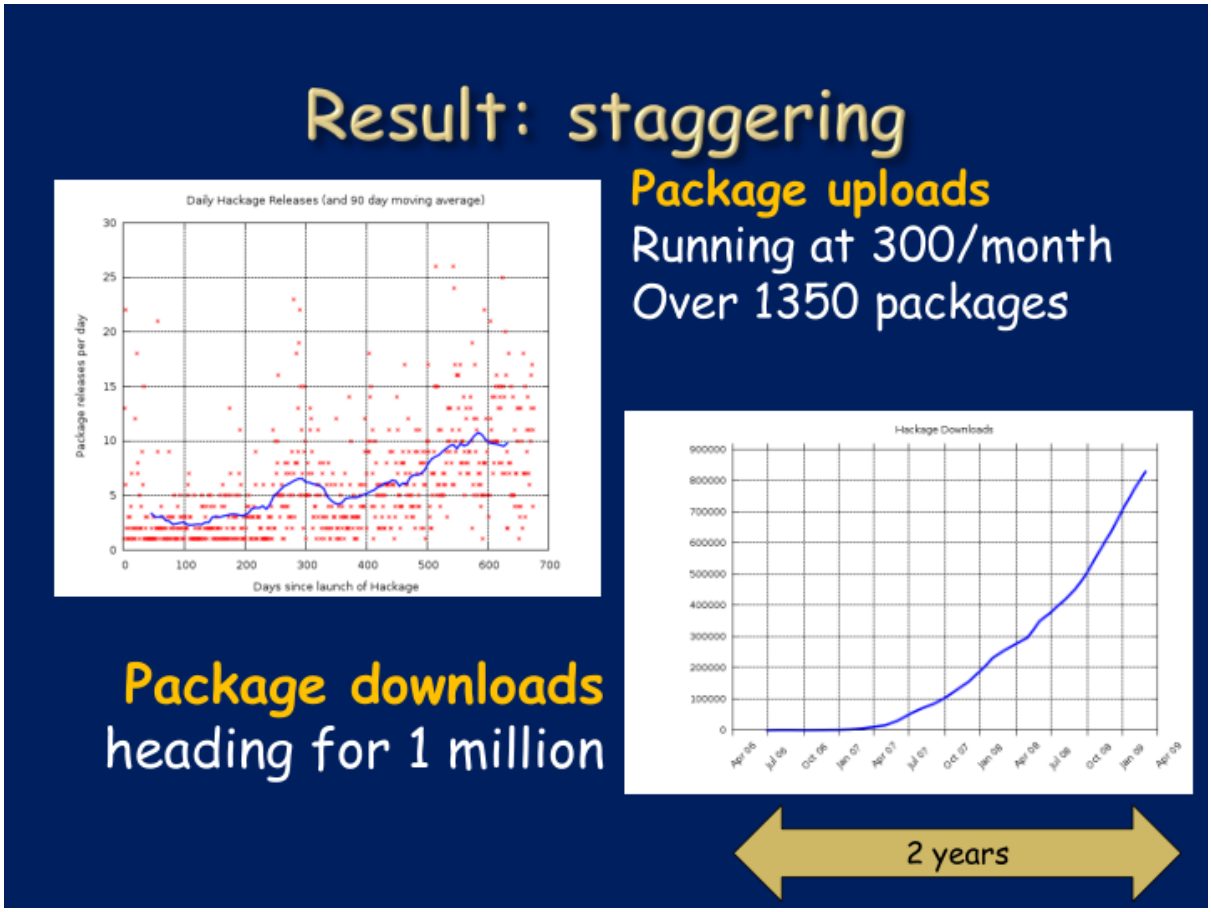

Рис. 12: вторая жизнь Haskell

- частичное применение функций и каррирование;
- ленивые (отложенные) вычисления (lazy evaluation);
- сопоставление с образцом (англ. pattern matching);
- недопустимость побочных эффектов, для ввода-вывода и ограниченного применения побочных эффектов используется концепция монад;
- статическая сильная типизация и автоматическое выведение типов (используется теорема Хиндли–Милнера);
- параметрический полиморфизм и др. виды полиморфизма (напр., ad hoc), мощная система классов типов;
- алгебраические типы данных (даже рекурсивные и псевдобесконечные);

ru.wikipedia: Объект первого класса

Обычно *объектами первого класса* (first-class object) называются элементы, которые могут быть переданы как параметр, возвращены из функции, присвоены переменной.

Недостатков у языка, разумеется, тоже хватает. Это и трудный порог вхождения для новичков, и непривычная для «императивщиков» функциональная парадигма, и сложности во взаимодействии с компилятором, и сложности с портабельностью на различные платформы (постепенно устраняется), и большой размер бинарников. Однако, отмеченные недостатки никак не умаляют достоинств языка.

Использование Haskell в IT-сфере

из более-менее известных «историй успеха»:

- Facebook использует для фильтрации спама Facebook's New Spam-Killer Hints at the Future of Coding;
- seL4, первое верифицированное микроядро, Haskell использовался для прототипирования при разработке и верификации;
- Xmonad — оконный менеджер для X Window System, полностью написан на Haskell;
- Pandoc — консольный мультиформатный конвертер документов, полностью написан на Haskell;
- Darcs — распределённая система управления версиями с широкими возможностями (напр., алгебра патчей);
- Linspire (дистрибутив Linux) использует Haskell как «системное средство» разработки
- GHC — один из самых мощных и успешных проектов: собственный компилятор (!!).

Haskell — специфика языка

Файлы по умолчанию имеют расширение «.hs».

строчные комментарии: в любой строке после двух дефисов `--` строка считается закомментированной.

многострочный комментарии: содержится между символами `{-` и `-}`.

https://wiki.haskell.org/Reference_card#Comments

https://wiki.haskell.org/Commenting

http://www.haskell.ru/lexemes.html#sect2.3

Литературный Haskell (lhs-формат)

Если комментарии начинают сильно «напрягать», Haskell предлагает особый формат с расширением «.lhs».

http://www.haskell.ru/literate.html#sect9.6

https://www.haskell.org/onlinereport/literate.html

Literate_programming

Соглашение о «грамотных комментариях», впервые разработанное Ричардом Бёрдом (Richard Bird) и Филиппом Уодлером (Philip Wadler) для Orwell, и позаимствованное в свою очередь Дональдом Кнутом (Donald Knuth) для «грамотного программирования», является альтернативным стилем программирования исходного кода на Haskell.

В этом случае, текст файла является комментарием по умолчанию.

Программный код вводится либо «птичьим стилем», т.е. значком «>» перед строками с кодом:

```
> fact :: Integer -> Integer
> fact 0 = 1
> fact n = n * fact (n-1)
```

(перед кодом и после него необходимо отступить пустую строку)

Либо, вводим более явно:

```
\begin{code}
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
\end{code}
```

Двумерный синтаксис

Мягкое введение в Haskell. Отбивка текста

haskell.ru: неформальные правила размещения текста программы

haskell.ru: правила размещения текста программы

Мелкие полезности

В первой, «студенческой», стадии программирования на Haskell несколько простых функций будет полезны:

```
myfoo :: Int -> Int -> Int
myfoo = undefined
```

Функция `myfoo` будет считаться определённой (как бы), её типом будет нами указанный тип, компиляция пройдёт успешно. Тем не менее, попытка исполнения даст ошибку времени выполнения:

```
*Main> myfoo 2 3
*** Exception: Prelude.undefined
CallStack (from HasCallStack):
  error, called at libraries\base\GHC\Err.hs:79:14 in base:GHC.Err
  undefined, called at undef.hs:2:9 in main:Main
```

И ещё для отладки иногда полезна функция `error`:

```
foo :: Double -> Double
foo x | x >= 0 = sqrt x
      | x <  0 = error "Use positive numbers!"
```

Однако, функцией `error` злоупотреблять не следует, её предназначение — работа с исключительными ситуациями. Для случаев возврата особых значений у нас будут специальные возможности.

Некоторые предопределенные типы в Haskell

см. Предопределенные типы и классы

Булевский: `Bool`

```
data  Bool  =  False | True deriving
                              (Read, Show, Eq, Ord, Enum, Bounded)
```

Основные булевские функции — это `&&` (и), `||` (или) и `not` (не).

Символы и строки: `Char`, `String`

`'a'`, `'я'` ...

`"Hello, всем!"`

Числа: `Int`, `Integer`, `Float`, `Double`.

Подробнее смотреть тут: Числа

Интересный комментарий на Habr.ru по поводу предопределённых типов:

> О, какой шикарный вопрос!
>
> Примитивные в смысле «не определяемые пользователем без адовых костылей» — Int, Integer, Double, Float, Char, вроде как ghc'шное расширение Word{8,16,32,64}. Если добавить полиморфные типы, то это ST, IO и тип функций (->), требующие поддержки компилятора и потому тоже примитивные. Если добавить то, что можно написать руками на чистом хаскеле, то ещё получим списки, туплы, всякие там Maybe, Either и прочее, но я, опять же, не уверен, что их стоит считать примитивными, несмотря на то, что они есть в стандарте языка. Если брать ghc'шные расширения (ибо нет компилятора, кроме ghc, и Саймон-Пейтон Джонс пророк его), то стоит упомянуть примитивные unboxed-типы вроде unboxed-целых, даблов,

флоатов, символов, туплов и так далее — их руками реализовать нельзя, они тоже требуют поддержку компилятора.

Определения функций в Haskell

Pattern Matching, guards

Pattern Matching и рекурсия:

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

ru.wikipedia: Сопоставление с образцом

без рекурсии:

```
f 3.14 = ”Pi!”
f _    =  ”другое число”
```

Символ «_» означает *любое значение подходящего типа*. Уравнения рассматриваются по порядку сверху вниз.

В следующем определении мы установим «охрану» (guards), т.е. зададим ряд условий, которые также будут просматриваться по порядку. Если условие `True`, то вычисляется выражение после равенства, если `False`, то просматривается следующее условие. При этом условия могут пересекаться или даже не покрывать все возможные ситуации. Если все условия будут просмотрены и ни одно из них не даст `True`, то получим ошибку времени выполнения.

Для получения сведений о типе функции, в интерпретаторе можно задать `:t fact'` или для более подробных сведений `:i fact'`.

Константа `otherwise` равна `True` и используется для удобства как синоним.

```
fact' x | x <= 1    = 1
        | otherwise = x * fact' (x-1)
```

Особым типом рекурсии является так называемая «ко-рекурсия». Так, если в рекурсивном определении мы используем значения функции с меньшим аргументом `(n-1)` обычно двигаясь к 0, то в определении функций с «ко-рекурсией» мы используем значения функции с большим аргументом `(n+1)`, двигаясь к ∞.

Допустим, у нас есть предикат `isprime`, который возвращает значение `True`, если целое число простое, и `False`, если нет. Тогда с помощью ко-рекурсии можем легко создать функцию поиска простого числа большего или равного заданному `x`:

```
nextprime x = if isprime x then x else nextprime (x+1)
```

(о ветвлении ниже)

Ветвления

В Haskell возможно описание вычисления функции в зависимости от условий в более-менее традиционном ключе, для чего присутствуют операторы **`if`** и **`case`**. Однако, их

применение отлично от аналогичных операторов в императивных языках. (Правильнее их называть выражениями с использованием **case**- и **if**-then-**else**-конструкций)

> Тем не менее, семантика данного оператора в Haskell похожа на семантику тернарного оператора
>
> ```
> y = x > 9 ? 100 : 200;
> ```
>
> в Си-подобных языках программирования.

Ключевое слово **if** предназначено для указание на ветвление вычислительного процесса в зависимости от условия булевского типа. Части then и **else** обязательны, они в отличие от императивного аналога задают не порядок действий, а функции, которые возвращают результат для задаваемой функции. Данное выражение является частным, более простым случаем применения выражений с **case**. Иными словами, условное выражение вида

```
nosign x = if x >= 0 then x else negate x
```

эквивалентно выражению

```
fcase x = case x >= 0 of
           True  -> x
           False -> negate x
```

Рассмотрим более сложные примеры с **case**-выражениями:

```
casing :: Integer -> Integer -> Integer
casing x y = case (x, y) of
               (_, 0) -> x
               (_, n) -> 1 + casing x (n-1)

casing' :: Integer -> Integer -> Integer
casing' x y = case (x, y) of
               (_, 0) | ( x >= 0 && y >= 0 ) -> x
               (_, y) | ( x >= 0 && y >= 0 ) -> 1 + casing' x (y-1)
               (_,_)  | ( x < 0 || y < 0   ) -> error
                                 ”cannot work with negative x and y”
```

Последний пример является комбинацией использования **case**-выражения с «охраной» для предотвращения применения функции для отрицательных аргументов.

Стоит также упомянуть, что задание функций уравнениями также возможно рассматривать как упрощённую запись с соответствующими **case**-выражениями.

if-then-else, case expressions, and guards! Oh my!