Функциональное программирование

Лекция 2

Степан Львович Кузнецов

НИУ ВШЭ, факультет компьютерных наук

Вычисление как преобразование

- В функциональной парадигме *вычисление* функции (программы) F на входных данных $a_1, \dots, a_n$ — это *редукция* (преобразование) терма $Fa_1 \dots a_n$ вплоть до *нормальной формы* — далее не редуцируемого состояния.

Вычисление как преобразование

- В функциональной парадигме *вычисление* функции (программы) F на входных данных $a_1, \dots, a_n$ — это *редукция* (преобразование) терма $Fa_1 \dots a_n$ вплоть до *нормальной формы* — далее не редуцируемого состояния.
- Базовый язык — «чистое» λ-исчисление, в котором термы строятся с помощью операций применения и λ-абстракции, а основное преобразование — β-редукция:

$$\boxed{\quad \dots \quad (\lambda x.u)v \quad \dots \quad} \to_\beta \boxed{\quad \dots \quad u[x := v] \quad \dots \quad}$$

Вычисление как преобразование

- В функциональной парадигме *вычисление* функции (программы) F на входных данных $a_1, \dots, a_n$ — это *редукция* (преобразование) терма $Fa_1 \dots a_n$ вплоть до *нормальной формы* — далее не редуцируемого состояния.
- Базовый язык — «чистое» λ-исчисление, в котором термы строятся с помощью операций применения и λ-абстракции, а основное преобразование — β-редукция:

$$\boxed{\quad \dots \quad (\lambda x.u)v \quad \dots \quad} \to_\beta \boxed{\quad \dots \quad u[x := v] \quad \dots \quad}$$

- Редукции могут применяться в разном порядке.

Порядок редукций

- Имеет место *свойство Чёрча – Россера:*

$$
\begin{tikzcd}
 & v_1 \arrow[dr, dashed, ->>] & \\
u \arrow[ur, ->>] \arrow[dr, ->>] & & w \\
 & v_2 \arrow[ur, dashed, ->>] &
\end{tikzcd}
$$

Порядок редукций

- Имеет место *свойство Чёрча – Россера:*

$$
\begin{tikzcd}
 & v_1 \arrow[dr, dashed, ->>] & \\
u \arrow[ur, ->>] \arrow[dr, ->>] & & w \\
 & v_2 \arrow[ur, dashed, ->>] &
\end{tikzcd}
$$

- Значит, совершить «ошибочную» редукцию на пути к нормальной форме невозможно: если нормальная форма существует, то любую стартовую последовательность редукций можно до неё довести.

Порядок редукций

- Имеет место *свойство Чёрча – Россера:*

$$
\begin{tikzcd}
 & v_1 \arrow[dr, dashed, two heads] & \\
u \arrow[ur, two heads] \arrow[dr, two heads] & & w \\
 & v_2 \arrow[ur, dashed, two heads] &
\end{tikzcd}
$$

- Значит, совершить «ошибочную» редукцию на пути к нормальной форме невозможно: если нормальная форма существует, то любую стартовую последовательность редукций можно до неё довести.
- В частности, нормальная форма, если существует, то α-единственна.

Порядок редукций

- Теорема Чёрча – Россера имеет место для «чистого» λ-исчисления, однако в его реально используемых расширениях может нарушаться.

Порядок редукций

- Теорема Чёрча – Россера имеет место для «чистого» λ-исчисления, однако в его реально используемых расширениях может нарушаться.
- Например, в Haskell'е есть `undefined` для аварийного прерывания вычисления.

Порядок редукций

- Теорема Чёрча – Россера имеет место для «чистого» λ-исчисления, однако в его реально используемых расширениях может нарушаться.
- Например, в Haskell'е есть `undefined` для аварийного прерывания вычисления.
- В присутствии `undefined` (мы его обозначим как $\bot$) порядок вычислений существен.

Порядок редукций

- Теорема Чёрча – Россера имеет место для «чистого» λ-исчисления, однако в его реально используемых расширениях может нарушаться.
- Например, в Haskell'е есть `undefined` для аварийного прерывания вычисления.
- В присутствии `undefined` (мы его обозначим как $\perp$) порядок вычислений существен.
- Пример: $(\lambda y.z)\perp$.

Порядок редукций

- Также бывают *слабо, но не сильно нормализуемые* термы, у которых есть нормальная форма, но есть и другой, бесконечный путь редукций.

Порядок редукций

- Также бывают *слабо, но не сильно нормализуемые* термы, у которых есть нормальная форма, но есть и другой, бесконечный путь редукций.
- Пример: $(\lambda y.z)\Omega$, где $\Omega = (\lambda x.(xx))(\lambda x.(xx))$.

Порядок редукций

- Также бывают *слабо, но не сильно нормализуемые* термы, у которых есть нормальная форма, но есть и другой, бесконечный путь редукций.
- Пример: $(\lambda y.z)\Omega$, где $\Omega = (\lambda x.(xx))(\lambda x.(xx))$.
- Поэтому, несмотря на свойство Чёрча – Россера, *порядок применения редукций важен.*

Порядок редукций

- Также бывают *слабо, но не сильно нормализуемые* термы, у которых есть нормальная форма, но есть и другой, бесконечный путь редукций.
- Пример: $(\lambda y.z)\Omega$, где $\Omega = (\lambda x.(xx))(\lambda x.(xx))$.
- Поэтому, несмотря на свойство Чёрча – Россера, *порядок применения редукций важен.*
- Традиционно (в императивных языках) используется *ретивый* (eager) порядок вычисления: сначала вычислить значения аргументов функции, потом саму функцию.

Порядок редукций

- Также бывают *слабо, но не сильно нормализуемые* термы, у которых есть нормальная форма, но есть и другой, бесконечный путь редукций.
- Пример: $(\lambda y.z)\Omega$, где $\Omega = (\lambda x.(xx))(\lambda x.(xx))$.
- Поэтому, несмотря на свойство Чёрча – Россера, *порядок применения редукций важен.*
- Традиционно (в императивных языках) используется *ретивый* (eager) порядок вычисления: сначала вычислить значения аргументов функции, потом саму функцию.
- В нашем примере $(\lambda y.z)\Omega$ такой порядок приводит к бесконечному циклу, пытаясь вычислить Ω.

Порядок вычислений

- Элементы других, *ленивых* (lazy) вычислений имеются и в некоторых императивных языках, например, в C:

```c
if (x != 0 && y/x > 3) { /* ... */ }
```

Порядок вычислений

- Элементы других, *ленивых* (lazy) вычислений имеются и в некоторых императивных языках, например, в C:

```c
if (x != 0 && y/x > 3) { /* ... */ }
```

  - Если x равно 0, то первый член конъюнкции ложен, значит, ложна и вся конъюнкция, и стандарт языка предписывает *не вычислять* второй член (что привело бы к ошибке «деление на ноль»).

Порядок вычислений

- Элементы других, *ленивых* (lazy) вычислений имеются и в некоторых императивных языках, например, в C:

```
if (x != 0 && y/x > 3) { /* ... */ }
```

  - Если x равно 0, то первый член конъюнкции ложен, значит, ложна и вся конъюнкция, и стандарт языка предписывает *не вычислять* второй член (что привело бы к ошибке «деление на ноль»).
- Мы определим *нормальную* стратегию редукций, соответствующую идее ленивого вычисления: не вычисляй значение, пока оно не понадобится.

Порядок вычислений

- Элементы других, *ленивых* (lazy) вычислений имеются и в некоторых императивных языках, например, в С:

```c
if (x != 0 && y/x > 3) { /* ... */ }
```

  - Если x равно 0, то первый член конъюнкции ложен, значит, ложна и вся конъюнкция, и стандарт языка предписывает *не вычислять* второй член (что привело бы к ошибке «деление на ноль»).
- Мы определим *нормальную* стратегию редукций, соответствующую идее ленивого вычисления: не вычисляй значение, пока оно не понадобится.
- Нормальная стратегия редукций реализует идею ленивости *последовательно.*

Ленивость

Для сравнения: при «традиционном» подходе, даже если реализация булевых операций ленивая, в более сложных случаях ленивость исчезает.

```
x = 0
y = 3

if (x == 0 or y/x > 3):
    print ”Hello!”

if ((lambda b: (x == 0 or b)) (y/x > 3)):
    print ”Hi!”
```

Нормальная стратегия редукций

- Говорим, что один редекс находится *левее* другого, если λ первого редекса расположена левее (в записи терма), чем λ второго.

Нормальная стратегия редукций

- Говорим, что один редекс находится *левее* другого, если λ первого редекса расположена левее (в записи терма), чем λ второго.
- Это означает, что либо первый редекс целиком расположен левее второго, либо второй редекс находится внутри первого (в u_1 или в v_1):

$$\ldots \quad (\lambda x.u_1)v_1 \quad \ldots \quad (\lambda x.u_2)v_2 \quad \ldots$$

$$\ldots \quad (\lambda x.\underbrace{\boxed{\ldots\ (\lambda x.u_2)v_2\ \ldots}}_{u_1})v_1 \quad \ldots$$

$$\ldots \quad (\lambda x.u_1)\underbrace{\boxed{\ldots\ (\lambda x.u_2)v_2\ \ldots}}_{v_1} \quad \ldots$$

Нормальная стратегия редукций

- **Нормальная стратегия:** всегда редуцируй *самый левый редекс.*

Нормальная стратегия редукций

- **Нормальная стратегия:** всегда редуцируй *самый левый редекс.*
- При этом это не обязательно самая левая λ: левее могут быть лямбды, не образующие β-редексов (после которых не идёт применение).

Нормальная стратегия редукций

- **Нормальная стратегия:** всегда редуцируй *самый левый редекс.*
- При этом это не обязательно самая левая λ: левее могут быть лямбды, не образующие β-редексов (после которых не идёт применение).
- В частности, мы сначала редуцируем $(\lambda x.u_1)v_1$, а только потом (если потребуется) вычисляем внутри v_1 (ленивость!).

Нормальная стратегия редукций

- **Нормальная стратегия:** всегда редуцируй *самый левый редекс.*
- При этом это не обязательно самая левая λ: левее могут быть лямбды, не образующие β-редексов (после которых не идёт применение).
- В частности, мы сначала редуцируем $(\lambda x.u_1)v_1$, а только потом (если потребуется) вычисляем внутри v_1 (ленивость!).

Теорема

Если терм можно привести к нормальной форме, то нормальная стратегия добьётся этого.

Вызов по необходимости

- Как мы уже видели, нормальная стратегия редукций избавляет от вычисления ненужных аргументов: $(\lambda xy.x)v_1v_2 \twoheadrightarrow_\beta v_1$.

Вызов по необходимости

- Как мы уже видели, нормальная стратегия редукций избавляет от вычисления ненужных аргументов: $(\lambda xy.x)v_1v_2 \twoheadrightarrow_\beta v_1$.
- С другой стороны, буквальное следование нормальной стратегии приводит к избыточным вычислениям за счёт копирования аргумента:

$$(\lambda x.\boxed{\ \dots\ x\ \dots\ x\ \dots\ x\ \dots\ })v \to_\beta \boxed{\ \dots\ v\ \dots\ v\ \dots\ v\ \dots\ }$$

Вызов по необходимости

- Как мы уже видели, нормальная стратегия редукций избавляет от вычисления ненужных аргументов: $(\lambda xy.x)v_1v_2 \twoheadrightarrow_\beta v_1$.
- С другой стороны, буквальное следование нормальной стратегии приводит к избыточным вычислениям за счёт копирования аргумента:

$$(\lambda x.\boxed{\ \dots\ x\ \dots\ x\ \dots\ x\ \dots\ })v \to_\beta \boxed{\ \dots\ v\ \dots\ v\ \dots\ v\ \dots\ }$$

- Для решения этой проблемы используется (в частности, в Haskell'e) *графовая оптимизация*, или «вызов по необходимости» (call-by-need):

Вызов по...

- **Вызов по значению** (call-by-value): сначала вычислить значения аргументов, потом применять функцию. Соответствует *аппликативному* порядку редукций, обычен для императивных языков.
- **Вызов по имени** (call-by-name): сначала подставить аргументы (не вычисляя их) в функцию, соответствует нормальному порядку редукций.
- **Вызов по необходимости** (call-by-need): соответствует порядку редукций с графовой оптимизацией.

Слабая головная нормальная форма

- Ещё один недостаток нормализации — её неустойчивость относительно расширения терма.

Слабая головная нормальная форма

- Ещё один недостаток нормализации — её неустойчивость относительно расширения терма.
- Например, терм $\lambda x.(x\Omega)$ не нормализуем (единственная редукция переводит Ω в себя), однако если рассмотреть его в большем терме: $(\lambda x.(x\Omega))(\lambda y.z)$, то этот терм нормализуется к z с помощью нормальной стратегии.

Слабая головная нормальная форма

- Ещё один недостаток нормализации — её неустойчивость относительно расширения терма.
- Например, терм $\lambda x.(x\Omega)$ не нормализуем (единственная редукция переводит Ω в себя), однако если рассмотреть его в большем терме: $(\lambda x.(x\Omega))(\lambda y.z)$, то этот терм нормализуется к z с помощью нормальной стратегии.
- Решение этой проблемы — отказ от применения некоторых редукций, т.е. ослабление требований к нормальной форме.

Слабая головная нормальная форма

- Ещё один недостаток нормализации — её неустойчивость относительно расширения терма.
- Например, терм $\lambda x.(x\Omega)$ не нормализуем (единственная редукция переводит Ω в себя), однако если рассмотреть его в большем терме: $(\lambda x.(x\Omega))(\lambda y.z)$, то этот терм нормализуется к z с помощью нормальной стратегии.
- Решение этой проблемы — отказ от применения некоторых редукций, т.е. ослабление требований к нормальной форме.
- Для этого используется *слабая головная нормальная форма* (WHNF), в которой разрешены редексы в определённых местах.

Слабая головная нормальная форма

- Ещё один недостаток нормализации — её неустойчивость относительно расширения терма.
- Например, терм $\lambda x.(x\Omega)$ не нормализуем (единственная редукция переводит Ω в себя), однако если рассмотреть его в большем терме: $(\lambda x.(x\Omega))(\lambda y.z)$, то этот терм нормализуется к z с помощью нормальной стратегии.
- Решение этой проблемы — отказ от применения некоторых редукций, т.е. ослабление требований к нормальной форме.
- Для этого используется *слабая головная нормальная форма* (WHNF), в которой разрешены редексы в определённых местах.
- Всякая нормальная форма является WHNF, но не наоборот.

Слабая головная нормальная форма

- В чистом λ-исчислении к термам в WHNF относятся:

Слабая головная нормальная форма

- В чистом λ-исчислении к термам в WHNF относятся:
    1. **все** термы вида $\lambda x.u$;

Слабая головная нормальная форма

- В чистом λ-исчислении к термам в WHNF относятся:
    1. **все** термы вида $\lambda x.u$;
    2. термы вида $x\, v_1 \; \ldots \; v_n$, где x — переменная. (При этом внутри v_i могут быть редексы.)

Слабая головная нормальная форма

- В чистом λ-исчислении к термам в WHNF относятся:
    1. **все** термы вида $\lambda x.u$;
    2. термы вида $x\, v_1 \ \dots\ v_n$, где x — переменная. (При этом внутри v_i могут быть редексы.)
- Таким образом, мы не вычисляем тогда, когда это может не пригодиться:

Слабая головная нормальная форма

- В чистом λ-исчислении к термам в WHNF относятся:
    1. **все** термы вида $\lambda x.u$;
    2. термы вида $x\, v_1\ \ldots\ v_n$, где x — переменная. (При этом внутри v_i могут быть редексы.)
- Таким образом, мы не вычисляем тогда, когда это может не пригодиться:
    1. функция с внешней λ'ой ещё не применена;

Слабая головная нормальная форма

- В чистом λ-исчислении к термам в WHNF относятся:
    1. **все** термы вида $\lambda x.u$;
    2. термы вида $x\, v_1\ \dots\ v_n$, где x — переменная. (При этом внутри v_i могут быть редексы.)
- Таким образом, мы не вычисляем тогда, когда это может не пригодиться:
    1. функция с внешней λ'ой ещё не применена;
    2. переменная x обозначает неизвестную функцию.

Слабая головная нормальная форма

- В чистом λ-исчислении к термам в WHNF относятся:
    1. **все** термы вида $\lambda x.u$;
    2. термы вида $x\, v_1\ \ldots\ v_n$, где x — переменная. (При этом внутри v_i могут быть редексы.)
- Таким образом, мы не вычисляем тогда, когда это может не пригодиться:
    1. функция с внешней λ'ой ещё не применена;
    2. переменная x обозначает неизвестную функцию.
- Недоредуцированные подтермы называются thunk'ами.

Слабая головная нормальная форма

- В чистом λ-исчислении к термам в WHNF относятся:
  1. **все** термы вида $\lambda x.u$;
  2. термы вида $x\, v_1\, \dots\, v_n$, где x — переменная. (При этом внутри v_i могут быть редексы.)
- Таким образом, мы не вычисляем тогда, когда это может не пригодиться:
  1. функция с внешней λ'ой ещё не применена;
  2. переменная x обозначает неизвестную функцию.
- Недоредуцированные подтермы называются thunk'ами.
- В Haskell'е, из-за другого синтаксиса, понятие WHNF немного другое (было/будет на семинаре).

Слабая головная нормальная форма

- К примеру, определим (в GHCi) ненормализуемый терм:

```
om = (let y = y in y)
```

Слабая головная нормальная форма

- К примеру, определим (в GHCi) ненормализуемый терм:

  ```
  om = (let y = y in y)
  ```

- Попытка вычислить `om` уводит в бесконечный цикл.

Слабая головная нормальная форма

- К примеру, определим (в GHCi) ненормализуемый терм:

  ```
  om = (let y = y in y)
  ```

- Попытка вычислить `om` уводит в бесконечный цикл.
- Однако если определить функцию

  ```
  kk = \x -> x om
  ```

  то попытка её вычислить даёт уже ошибку “no instance for Show” — т.е. `om` здесь не пытаются вычислить.

Слабая головная нормальная форма

- К примеру, определим (в GHCi) ненормализуемый терм:

  ```
  om = (let y = y in y)
  ```

- Попытка вычислить `om` уводит в бесконечный цикл.
- Однако если определить функцию

  ```
  kk = \x -> x om
  ```

  то попытка её вычислить даёт уже ошибку “no instance for Show” — т.е. `om` здесь не пытаются вычислить.
- В `kk (\z -> z)`, конечно, будет бесконечный цикл, а вот `kk (\z -> 0)` лениво вычисляется в `0`.

Проблемы с ленивостью

Пример из `https://eax.me/lazy-evaluation/` „Скандальная правда о Haskell и ленивых вычислениях“

- Иногда стратегия вызова по необходимости, несмотря на графовую оптимизацию, приводит к нежелательным с точки зрения эффективности последствиям.

Проблемы с ленивостью

Пример из `https://eax.me/lazy-evaluation/` „Скандальная правда о Haskell и ленивых вычислениях“

- Иногда стратегия вызова по необходимости, несмотря на графовую оптимизацию, приводит к нежелательным с точки зрения эффективности последствиям.
- Рассмотрим следующий пример (вычисление суммы элементов списка):

```haskell
mysum x = mysum' 0 x
mysum' acc [] = acc
mysum' acc (x:xs) = mysum' (acc+x) xs
main = putStrLn (show (mysum [1..1000000]))
```

Проблемы с ленивостью

- Эта программа выдаёт правильный ответ (500000500000).

Проблемы с ленивостью

- Эта программа выдаёт правильный ответ (500000500000).
- Посмотрим, однако, на использование ресурсов.

```
ghc -rtsopts lazy_fail.hs
./lazy_fail +RTS -sstderr
```

Проблемы с ленивостью

- Эта программа выдаёт правильный ответ (500000500000).
- Посмотрим, однако, на использование ресурсов.

  ```
  ghc -rtsopts lazy_fail.hs
  ./lazy_fail +RTS -sstderr
  ```
- Получаем “`99 MiB total memory in use`” (и это число будет меняться в зависимости от размера массива).

Проблемы с ленивостью

- Эта программа выдаёт правильный ответ (500000500000).
- Посмотрим, однако, на использование ресурсов.

  ```
  ghc -rtsopts lazy_fail.hs
  ./lazy_fail +RTS -sstderr
  ```
- Получаем “`99 MiB total memory in use`” (и это число будет меняться в зависимости от размера массива).
- Проблема не в глубине стека: `mysum'` реализован через хвостовую рекурсию, она оптимизируется.

Проблемы с ленивостью

- Эта программа выдаёт правильный ответ (500000500000).
- Посмотрим, однако, на использование ресурсов.

```
ghc -rtsopts lazy_fail.hs
./lazy_fail +RTS -sstderr
```

- Получаем “`99 MiB total memory in use`” (и это число будет меняться в зависимости от размера массива).
- Проблема не в глубине стека: `mysum'` реализован через хвостовую рекурсию, она оптимизируется.
- Дело в порядке редукций и слишком больших thunk’ах.

Проблемы с ленивостью

- Последовательность редукций:
  `mysum' 0 [0..3]` → `mysum' (0+0) [1..3]` →
  `mysum' (0+0+1) [2..3]` → `mysum' (0+0+1+2) [3]` →
  `mysum' (0+0+1+2+3) []` ↠ 6

Проблемы с ленивостью

- Последовательность редукций:
  `mysum' 0 [0..3]` → `mysum' (0+0) [1..3]` →
  `mysum' (0+0+1) [2..3]` → `mysum' (0+0+1+2) [3]` →
  `mysum' (0+0+1+2+3) []` ↠ 6
- Аккумулятор acc в процессе вычислений остаётся огромным thunk'ом, а фактически вычисляется только в самом конце.

Проблемы с ленивостью

- Последовательность редукций:
  `mysum' 0 [0..3]` → `mysum' (0+0) [1..3]` → `mysum' (0+0+1) [2..3]` → `mysum' (0+0+1+2) [3]` → `mysum' (0+0+1+2+3) []` ↠ `6`
- Аккумулятор acc в процессе вычислений остаётся огромным thunk'ом, а фактически вычисляется только в самом конце.
- Получается, что мы храним наш большой массив `[1..1000000]` не в компактном, а в явном виде.

Проблемы с ленивостью

- Последовательность редукций:
  `mysum' 0 [0..3]` → `mysum' (0+0) [1..3]` →
  `mysum' (0+0+1) [2..3]` → `mysum' (0+0+1+2) [3]` →
  `mysum' (0+0+1+2+3) []` ↠ 6
- Аккумулятор acc в процессе вычислений остаётся огромным thunk'ом, а фактически вычисляется только в самом конце.
- Получается, что мы храним наш большой массив `[1..1000000]` не в компактном, а в явном виде.
- Чтобы избежать этого, нужно принудить Haskell сразу вычислять (приводить к WHNF) выражение acc+x.

Проблемы с ленивостью

- Последовательность редукций:
  `mysum' 0 [0..3]` → `mysum' (0+0) [1..3]` →
  `mysum' (0+0+1) [2..3]` → `mysum' (0+0+1+2) [3]` →
  `mysum' (0+0+1+2+3) []` ↠ `6`
- Аккумулятор acc в процессе вычислений остаётся огромным thunk'ом, а фактически вычисляется только в самом конце.
- Получается, что мы храним наш большой массив `[1..1000000]` не в компактном, а в явном виде.
- Чтобы избежать этого, нужно принудить Haskell сразу вычислять (приводить к WHNF) выражение acc+x.
- Для этого используется встроенная функция seq.

Проблемы с ленивостью

- `seq` вычисляет свой первый аргумент и (если вычисление успешно) игнорирует его и возвращает второй.

Проблемы с ленивостью

- seq вычисляет свой первый аргумент и (если вычисление успешно) игнорирует его и возвращает второй.
- В нашем примере:

```
mysum x = mysum' 0 x
mysum' acc [] = acc
mysum' acc (x:xs) = (acc+x) `seq` mysum' (acc+x) xs
main = putStrLn (show (mysum [1..1000000]))
```

Проблемы с ленивостью

- seq вычисляет свой первый аргумент и (если вычисление успешно) игнорирует его и возвращает второй.
- В нашем примере:

```
mysum x = mysum' 0 x
mysum' acc [] = acc
mysum' acc (x:xs) = (acc+x) `seq` mysum' (acc+x) xs
main = putStrLn (show (mysum [1..1000000]))
```

- Здесь новое значение аккумулятора оказывается предвычисленным и (за счёт графовой оптимизации) именно оно передаётся по рекурсии.

Проблемы с ленивостью

- `seq` вычисляет свой первый аргумент и (если вычисление успешно) игнорирует его и возвращает второй.
- В нашем примере:

```
mysum x = mysum' 0 x
mysum' acc [] = acc
mysum' acc (x:xs) = (acc+x) `seq` mysum' (acc+x) xs
main = putStrLn (show (mysum [1..1000000]))
```

- Здесь новое значение аккумулятора оказывается предвычисленным и (за счёт графовой оптимизации) именно оно передаётся по рекурсии.
- Расход памяти — 2 MiB (столько же, сколько у тривиальной “Hello, World!”), и он не растёт с ростом длины списка.

Проблемы с ленивостью

- seq вычисляет свой первый аргумент и (если вычисление успешно) игнорирует его и возвращает второй.
- В нашем примере:

```
mysum x = mysum' 0 x
mysum' acc [] = acc
mysum' acc (x:xs) = (acc+x) `seq` mysum' (acc+x) xs
main = putStrLn (show (mysum [1..1000000]))
```

- Здесь новое значение аккумулятора оказывается предвычисленным и (за счёт графовой оптимизации) именно оно передаётся по рекурсии.
- Расход памяти — 2 MiB (столько же, сколько у тривиальной “Hello, World!”), и он не растёт с ростом длины списка.
- Через seq определяется оператор `f $! x`, который означает `` x `seq` (f x) ``

Изменение порядка редукций

- Итак, `seq` *изменяет порядок редукций.*

Изменение порядка редукций

- Итак, `seq` *изменяет порядок редукций.*
- На одной из следующих лекций мы познакомимся с ещё одним таким оператором — `par`, реализующим распараллеливание.

Типы в языках программирования

- В большинстве языков программирования имеются системы *типов данных.* Бестиповые языки, такие как языки ассемблера или простейшее λ-исчисление, встречаются редко.

Типы в языках программирования

- В большинстве языков программирования имеются системы *типов данных*. Бестиповые языки, такие как языки ассемблера или простейшее λ-исчисление, встречаются редко.
- В бестиповом языке любую операцию можно совершить над любыми данными. Дисциплина типов данных налагает определённые *ограничения* на применение операций (функций), чтобы отсечь *бессмысленные* ошибочные применения.

Типы в языках программирования

- В большинстве языков программирования имеются системы *типов данных.* Бестиповые языки, такие как языки ассемблера или простейшее λ-исчисление, встречаются редко.
- В бестиповом языке любую операцию можно совершить над любыми данными. Дисциплина типов данных налагает определённые *ограничения* на применение операций (функций), чтобы отсечь *бессмысленные* ошибочные применения.
  - Например, выражение `2+2` осмысленно (хотя, может быть, вычисляет не то, что нам на самом деле нужно), а выражение `2+”two”` скорее всего бессмысленно.

Типы в языках программирования

- Таким образом, система типов выполняет охранительную функцию: проверки корректности типов запрещают некоторые конструкции («мешают программировать»).

Типы в языках программирования

- Таким образом, система типов выполняет охранительную функцию: проверки корректности типов запрещают некоторые конструкции («мешают программировать»).
- При этом эти конструкции не всегда совершенно бессмысленные. Например, комбинатор неподвижной точки $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$ скорее всего будет некорректен с точки зрения системы типов (аргумент функции не может иметь тот же тип, что и сама функция), однако разумно используется для реализации рекурсии.

Типы в языках программирования

- Таким образом, система типов выполняет охранительную функцию: проверки корректности типов запрещают некоторые конструкции («мешают программировать»).
- При этом эти конструкции не всегда совершенно бессмысленные. Например, комбинатор неподвижной точки $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$ скорее всего будет некорректен с точки зрения системы типов (аргумент функции не может иметь тот же тип, что и сама функция), однако разумно используется для реализации рекурсии.
- Для собственно исполнения программы (вычисления) типы обыкновенно не нужны.

Типы в языках программирования

- С другой стороны, контроль типов помогает избежать многих ошибок при программировании.

Типы в языках программирования

- С другой стороны, контроль типов помогает избежать многих ошибок при программировании.
    - Фактически, контроль типов — это начальный элемент *верификации* (формального доказательства) корректности работы программы.

Типы в языках программирования

- С другой стороны, контроль типов помогает избежать многих ошибок при программировании.
  - Фактически, контроль типов — это начальный элемент *верификации* (формального доказательства) корректности работы программы.
  - Используя развитую систему типов *(зависимые типы)*, можно свести задачу верификации к проверке типов. Например, вместо $\mathrm{mod} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ можно потребовать более точный тип

$$\begin{aligned}\mathrm{mod}' : ((x, y) : \mathbb{N} \times \mathbb{N}) \mapsto \\ \mapsto r : \{r : \mathbb{N} \mid y = 0 \vee \exists q : \mathbb{N}(x = y \cdot q + r \wedge r < y)\},\end{aligned}$$

Типы в языках программирования

- С другой стороны, контроль типов помогает избежать многих ошибок при программировании.
  - Фактически, контроль типов — это начальный элемент *верификации* (формального доказательства) корректности работы программы.
  - Используя развитую систему типов *(зависимые типы)*, можно свести задачу верификации к проверке типов. Например, вместо $\mathrm{mod} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ можно потребовать более точный тип

$$\begin{aligned} \mathrm{mod}' : ((x, y) : \mathbb{N} \times \mathbb{N}) \mapsto \\ \mapsto r : \{r : \mathbb{N} \mid y = 0 \vee \exists q : \mathbb{N}(x = y \cdot q + r \wedge r < y)\}, \end{aligned}$$

  - Такие возможности есть в Coq, Agda и проч.

Типы в языках программирования

- Типы также используются как косвенный способ документирования программного кода: по типу функции зачастую можно понять, что она делает.

Типы в языках программирования

- Типы также используются как косвенный способ документирования программного кода: по типу функции зачастую можно понять, что она делает.
    - Например, из типа $\mathbf{B} : (B \to C) \to ((A \to B) \to (A \to C))$ даже без реализации ($\mathbf{B} = \lambda fgx.f(gx)$) понятно, что **B** реализует композицию функций.

Типы в языках программирования

- Типы также используются как косвенный способ документирования программного кода: по типу функции зачастую можно понять, что она делает.
  - Например, из типа $\mathbf{B} : (B \to C) \to ((A \to B) \to (A \to C))$ даже без реализации ($\mathbf{B} = \lambda fgx.f(gx)$) понятно, что $\mathbf{B}$ реализует композицию функций.
  - Более того, если это полиморфный тип, где A, B, C — абстрактные переменные, то можно *доказать*, что $\mathbf{B}$ — это оператор композиции. Это одна из так называемых *free theorems*.

Типы в языках программирования

- Типы также используются как косвенный способ документирования программного кода: по типу функции зачастую можно понять, что она делает.
    - Например, из типа $\mathbf{B} : (B \to C) \to ((A \to B) \to (A \to C))$ даже без реализации ($\mathbf{B} = \lambda fgx.f(gx)$) понятно, что **B** реализует композицию функций.
    - Более того, если это полиморфный тип, где A, B, C — абстрактные переменные, то можно *доказать*, что **B** — это оператор композиции. Это одна из так называемых *free theorems.*
- Наконец, типы влияют на исполнение кода при так называемом *ad hoc полиморфизме,* или *перегрузке* функции. Пример (работает в C++, но не в C):

```
void f(int x) { printf("integer\n"); }
void f(char x) { printf("character\n"); }
```