Система типов в Haskell

Система типов в языках программирования

Haskell является статически типизированным языком, более того система типов — *полная, сильная, статическая, с автоматическим выводом типов*, основанная на системе типов Хиндли–Милнера. Таким образом, она может быть еще и *неявная*.

Полная…

Стиль программирования, отличающийся обширным использованием информации о типах с тем, чтобы механизм проверки согласования типов обеспечил раннее выявление максимального количества всевозможных разновидностей багов. Полнотиповое программирование может поддерживаться на уровне системы типов языка или вводиться программистом идиоматически. Понятие ввёл Лука Карделли (eng. Luca Cardelli) в 1991 г. в одноимённой работе.

wikipedia: Полнотиповое программирование

Язык Haskell относят к полнотиповым.

Сильная…

(eng. strong typing и weak typing, также иногда говорят строгая/нестрогая)

Сильная типизация выделяется тем, что язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования, например нельзя вычесть из строки множество. Языки со слабой типизацией выполняют множество неявных преобразований автоматически, даже если может произойти потеря точности или преобразование неоднозначно.

Примеры:
Сильная: Java, Python, Haskell, Lisp; Слабая: C, JavaScript, Visual Basic, PHP.

Часто заявляют, что если язык статически типизирован, то вы сможете отловить множество потенциальных ошибок при компиляции. Это как раз не так, язык должен быть ещё и сильно типизирован.

> Понятия «сильный» и «слабый» — очень неоднозначные. Вот некоторые примеры их использования:
>
> Иногда «сильный» означает «статический». Тут все просто, но лучше использовать термин «статический», потому что большинство используют и понимают его.
>
> Иногда «сильный» означает «не делает неявное преобразование типов». Например, JavaScript позволяет написать `"a"+ 1`, что можно назвать «слабой типизацией». Но почти все языки предоставляют тот или иной уровень неявного преобразования, которое позволяет автоматически переходить от целых чисел к числам с плавающей запятой вроде `1 + 1.1`. В реальности, большинство людей используют слово «сильный» для определения границы между приемлемым и неприемлемым преобразованием. Нет какой-то общепринятой границы, они все неточные и зависят от мнения конкретного человека.

> Иногда “сильный” означает, что невозможно обойти строгие правила типизации в языке.
>
> Иногда “сильный” означает безопасный для памяти (memory-safe). Си — это пример небезопасного для памяти языка. Если `xs` — это массив четырех чисел, то Си с радостью выполнит код `xs[5]` или `xs[1000]`, возвращая какое-то значение из памяти, которая находится сразу за `xs`.

habr: Статическая и динамическая типизация

Haskell тоже умеет складывать: `1 + 1.1`, но тонкость тут заключается в том, что числовые литералы в Haskell — это перегруженные нуль-местные функции, принадлежащие широкому классу типов **`Num`** и тип этих литералов выводится в зависимости от места использования.

А вот если мы сделаем следующее:

```
>let x=2::Int; y=2::Double
> x+y
:2:3: error:
    * Couldn't match expected type `Int' with actual type `Double'
    * In the second argument of `(+)', namely `y'
      In the expression: x + y
      In an equation for `it': it = x + y
```

В языках подобных Си это соответствовало бы объявлению и инициализации переменных типа **`Int`** и **`Double`**, а потом совместного их использования в одном арифметическом выражении. Здесь мы вынуждены делать приведение типов в явном виде:

```
> fromIntegral(x) + y
4.0
```

wikipedia: Сильная и слабая типизация

Каламбур типизации

wiki.haskell: Strong typing

Статическая…

Главное, что отличает статическую (static) типизацию от динамической (dynamic) это то, что все проверки типов выполняются на этапе компиляции, а не на этапе выполнения.

Преимущества статической типизации:

- Проверки типов происходят только один раз — на этапе компиляции. А это значит, что нам не нужно будет постоянно выяснять, не пытаемся ли мы поделить число на строку (и либо выдать ошибку, либо осуществить преобразование).
- Скорость выполнения. Из предыдущего пункта ясно, что статически типизированные языки практически всегда быстрее динамически типизированных.
- При некоторых дополнительных условиях, позволяет обнаруживать потенциальные ошибки уже на этапе компиляции.

- Ускорение разработки при поддержке IDE (отсеивание вариантов, заведомо не подходящих по типу).

Преимущества динамической типизации:

- Простота создания универсальных коллекций — куч всего и вся (редко возникает такая необходимость, но когда возникает, динамическая типизация выручит).
- Удобство описания обобщенных алгоритмов (например сортировка массива, которая будет работать не только на списке целых чисел, но и на списке вещественных и даже на списке строк).
- Легкость в освоении — языки с динамической типизацией обычно очень хороши для того, чтобы начать программировать.

Самый важный аргумент за динамическую типизацию — удобство описания обобщенных алгоритмов. Давайте представим себе проблему: нам нужна функция поиска по разнообразным массивам (или спискам) — по массиву целых чисел, по массиву вещественных и массиву символов.

Алгоритм поиска — перебор. Функция будет получать искомый элемент, сам массив (или список) и возвращать индекс элемента, или, если элемент не найден: `(-1)`.

Динамическое решение (Python):

```
def find( required_element, list ):
    for (index, element) in enumerate(list):
        if element == required_element:
            return index

    return (-1)
```

Я ещё добавил рассмотрение примеров на Perl и Javascript, но в силу разных подходов с неявным приведением типов и особенностей работы в этих ситуациях для данных языков (напр., в Perl есть специальный оператор сравнения `eq`, а в Javascript помимо сравнения `===` есть ещё множества правил приведения) могут возникнуть проблемы при использовании смешанных массивов со строками и числами одновременно.

Perl:

```
sub find {
  my($e, @list) = @_;
  for ($i=0; $i<$#list; ++$i) {
    return $i if $e == $list[$i]
  };
  return (-1);
}
```

Javascript:

```
function find (e,list) {
  for (i=0; i < list.length; ++i) {
        if (e == list[i]) {return i};
  }
```

```
  return (-1);
}
```

Статическое решение (Си):

```
unsigned int find_int( int required_element,
  int array[], unsigned int size ) {
    for (unsigned int i = 0; i < size; ++i )
        if (required_element == array[i])
            return i;

    return (-1);
}

unsigned int find_float( float required_element,
  float array[], unsigned int size ) {
    for (unsigned int i = 0; i < size; ++i )
        if (required_element == array[i])
            return i;

    return (-1);
}

unsigned int find_char( char required_element,
  char array[], unsigned int size ) {
    for (unsigned int i = 0; i < size; ++i )
        if (required_element == array[i])
            return i;

    return (-1);
}
```

Отметим, что во взятом примере для Си из статьи про типы есть проблема — построенная функция **find** вместо типа `unsigned int` возвращает тип со знаком. Пример будет работать, тип будет преобразован к нужному, но ситуация показательна!

Статическое решение (обобщенное программирование, C++):

```
template <class T>
unsigned int find( T required_element,
  std::vector<T> array ) {
    for (unsigned int i = 0; i < array.size(); ++i )
        if (required_element == array[i])
            return i;

    return (-1);
}
```

А вот Haskell, являясь статическим, легко предоставляет возможность сделать обобщённое решение:

```
find x lst  = find' lst 0
  where
```

```
    find' [] _ = (-1)
    find' (y:ys) t = if x == y
                     then t
                     else find' (ys) (t+1)
```

Немного «сложноватый» код из-за необходимости поддерживать индексы у списка и рекурсивный разбор. Тем не менее, его полиморфный характер позволяет работать со списками любого рода элементов, которые поддерживают сравнения на равенство:

```
> :t find
find :: (Num t, Eq t1) => t1 -> [t1] -> t
```

Если надо совсем декларативно, то имеется готовое решение:

```
:m Data.List

> elemIndex 2 [0..5]
Just 2
```

Кстати, Haskell при необходимости может поддерживать и динамическую типизацию — модуль `Data.Dynamic` даёт такую возможность.

> Компиляторы статических языков обычно могут генерировать более быстрый код, чем компиляторы динамических. Например, если компилятор знает, что функция `add` принимает целые числа, то он может использовать нативную инструкцию `ADD` центрального процессора. Динамический язык будет проверять тип при выполнении, выбирая одну из множества функций `add` в зависимости от типов (складываем `integers` или `floats` или склеиваем строки или, может быть, списки?) Или нужно решить, что возникла ошибка и типы не соответствуют друг другу. Все эти проверки занимают время. В динамических языках используются разные трюки для оптимизации, например `JIT`-компиляция (`just-in-time`), где код перекомпилируется при выполнении после получения всей необходимой о типах информации. Однако, никакой динамический язык не может сравниться по скоростью с аккуратно написанным статическим кодом на языке вроде `Rust`.

Компиляция статически типизированного кода

Другие ссылки по теме:

Ликбез по типизации в языках программирования

Статическая и динамическая типизация

wikipedia: Теория типов

progopedia: Теория типов

Просто о Хиндли-Милнере

Отметим, что в языках с динамической типизацией возможны присваивания одной переменной значений разных типов в одной программе. Вот пример на языке с сильной динамической типизацией, python:

```
x = "hello"
```

```
print(x)
x = 1
print(x)
```

Явная и неявная

Язык с явной типизацией предполагает, что программист должен указывать типы всех переменных и функций, которые объявляет (eng.: explicit typing).

Язык с неявной типизацией, напротив, предлагает Вам забыть о типах и переложить задачу вывода типов на компилятор или интерпретатор (eng.: implicit typing).

Поначалу можно решить, что неявная типизация равносильна динамической, а явная — статической, но дальше мы увидим, что это не так.

Преимущества явной типизации:

- Наличие у каждой функции сигнатуры (например `int add(int, int)`) позволяет без проблем определить, что функция делает.
- Программист сразу записывает, какого типа значения могут храниться в конкретной переменной, что снимает необходимость запоминать это.

Преимущества неявной типизации

- Сокращение записи `def add(x, y)` явно короче, чем `int add( int x, int y)`.
- Устойчивость к изменениям. Например если в функции временная переменная была того же типа, что и у аргумента, то в явно типизированном языке при изменении типа аргумента функции нужно будет изменить еще и тип временной переменной.

Ряд современных языков позволяет пользоваться неявной типизацией по умолчанию и явной при необходимости. Haskell как раз один из них. Но если обычно языки используют так называемую утиную типизацию (Perl, Smalltalk, Python, Ruby, JavaScript, Lua, Go, Scala, etc..), то Haskell использует надёжные математические теоремы, и все это выполняется во время компиляции.

Рассмотрим пример.

```
-- Без явного указания типа
add1 (x, y) = x + y

-- Явное указание типа
add2 :: (Int, Int) -> Int
add2 (x, y) = x + y
```

У функции `add1` компилятор неявно (но математически точно) выведет наиболее общей числовой тип, в данном случае это будут вообще все типы, поддерживающие сложение, т.е.

```
add1 :: (Num a) => a -> a -> a
```

А у функции `add2` мы сами, явно задав тип, можем сделать её использование более оптимальным.

Haskell-программисты обычно использую явное описание типов как своего рода документацию своего кода.

Сравнительная таблица языков программирования

Это примерная таблица, и её содержание во многом субъективно, как было указано выше.

JavaScript	Динамическая	Слабая	Неявная
Ruby	Динамическая	Сильная	Неявная
Python	Динамическая	Сильная	Неявная
Java	Статическая	Сильная	Явная
PHP	Динамическая	Слабая	Неявная
C	Статическая	Слабая	Явная
C++	Статическая	Слабая	Явная
Perl	Динамическая	Слабая	Неявная
Objective-C	Статическая	Слабая	Явная
C#	Статическая	Сильная	Явная
Haskell	Статическая	Сильная	Неявная
Common Lisp	Динамическая	Сильная	Неявная
D	Статическая	Сильная	Явная
Delphi	Статическая	Сильная	Явная
Go	Статическая	Сильная	Неявная

Для более корректного сравнения, мы должны, конечно, пристально рассматривать языки. Так, в таблице языки Haskell и Go обладают одинаковыми записями. Однако, это сильно разные языки в возможностях системы типов. В системе типизации Go нет обобщённых типов, типов с «параметрами» от других типов.

Допустим, нам нужен свой тип `MyList` для списков, в котором можно хранить нужные нам данные. Мы хотим иметь возможность создавать `MyList` целых чисел, `MyList` строк и так далее, не меняя исходный код `MyList`. Компилятор должен следить за типизацией: если есть `MyList` целых чисел, и мы случайно добавляем туда строку, то компилятор должен отклонить программу.

Go специально был спроектирован таким образом, чтобы невозможно было задавать типы вроде `MyList`. Лучшее, что возможно сделать, это создать `MyList` «пустых интерфейсов»: `MyList` может содержать объекты, но компилятор просто не знает их тип. Когда мы достаем объекты из `MyList`, нам нужно сообщить компилятору их тип. Если мы говорим «Я достаю строку», но в реальности значение — это число, то будет ошибка исполнения, как в случае с динамическими языками.

Теперь давайте сравним с Haskell, который обладает очень мощной системой типов. Если задать тип `MyList`, то тип «списка чисел» это просто `MyList Integer`. Haskell не даст нам случайно добавить строку в список, и удостоверится, что мы не положим элемент из списка в строковую переменную.

Разнообразие статических систем типизации

Но Haskell может значительно больше! Можно написать функцию (типа сложения), которая будет работать с числами разных типов, строками, матрицами и т.п., но при этом следить, чтобы они не смешивались в одну кучу при вызове.

Система типов Haskell может помочь нам ещё на этапе компиляции отслеживать чистоту функции. Так, только по сигнатуре функции вида `Int -> String` мы понимаем, что эта функция чистая, а по сигнатуре вида `Int -> IO String` мы понимаем, что она осуществляет какой-то вывод строки во внешнее устройство.

Upd. Следует отметить, ситуация с языками меняется: и Java уже имеет вывод типов в некотором виде, и уже Golang получил дженерики.

Я до последнего буду защищать сильную статическую типизацию

Java Type Inference

Local Variable Type Inference (var)

Дженерики в Go — подробности из блога разработчиков

Общее представление о системе типов в Haskell

Алгебраические типы данных

Типы в Haskell бывают примитивные (базовые, как правило, глубоко встроенные в недра компилятора, хотя и могут быть представлены в педагогических целях тоже как алгебраические) и *алгебраические*. Алгебраическими они называются потому, что используют математические (алгебраические) понятия при конструировании: декартово произведение и прямую сумму (дизъюнктное объединение).

(некоторые примитивные типы данных, таких как `Bool`, `Char` можно представить как алгебраические)

История использования алгебраических типов началась не с Haskell (Hope, ML, OCaml) и им не закончилась (F#, Scala, Rust, Nemerle). Но с Haskell они стали наиболее популярны.

Итак, в Haskell программисту доступно конструирование именно алгебраических типов данных. Рассмотрим подробнее, как это делается.

wikipedia: Алгебраический_тип_данных

wikipedia: Обобщённый_алгебраический_тип_данных (GADT)

wiki.haskell: Algebraic data type

Р. Душкин. Алгебраические типы данных и их использование в программировании

Шикарный пост о «примитивных» типах в Haskell

Перечисления

Наиболее простой способ определения типов — это перечисления. Так, мы можем определить, например цветовой тип:

```
data Color = Red | Green | Blue | Black | White
```

Это как раз то, что алгебраически соответствовало *прямым суммам* (объединению непересекающихся множеств, например)

Однако, без представления о классах (см. далее), с данным типом можно работать лишь в сопоставлениях с образцом в определениях функций:

```
f :: Color -> Char
f Red     = 'r'
f Green   = 'g'
f Blue    = 'b'
f _ = 'x'
```

Необходимую терминологию мы введем в следующем разделе.

Типы произведений и записи

Другой, более интересный способ, так называемый *тип произведений* (нам это будет легче понимать как записи или структуры):

```
data Point a = Pt a a
```

Здесь `Point` называют *конструктором типов*, производящим тип, а `Pt` называют *конструктором данных* (или *конструктором значений*), производящим значение.

Значения `a` в `Pt a a` называются *полями*, они не обязаны быть одинаковыми, но обязаны быть одного типа.

> Как указывают авторы «Мягкого введения в Haskell», важно различать применение конструкторов типа и данных. Второе происходит во время выполнения и представляет собой способ, которым осуществляются вычисления в Haskell. Первое происходит во время компиляции и является частью процесса по обеспечению типобезопастности.
>
> Кроме того, конструкторы типа и данных находятся в разных пространствах имен. Таким образом, мы могли именем `Point` называть оба конструктора — так часто программисты на Haskell и поступают.

```
data Point a = Point a a
```

Алгебраические типы данных также возможно использовать в сопоставлениях с образцом:

```
fliP :: Point a -> Point a
fliP (Pt x y) = Pt y x

paiR :: Point t -> (t, t)
paiR (Pt x y) = (x,y)

getX :: Point t -> t
getX (Pt x _) = x
```

Такая техника работы достигается за счет _иммутабельности структур данных (созданных и с помощью АТД, и в целом) в Haskell. Таким образом, если мы создали некоторое значение (объект) с помощью конструктора значений, то этот неизменный объект будет всегда хранить информацию о том, как он был создан, и при необходимости, мы можем выполнить деконструкцию на исходные части в момент сравнения с образцом.

Возможно применение введённого конструктора типа с конкретными типами:

```
f :: Point Double -> Double
f (Pt x y) = sqrt (x^2 + y^2)

g :: Point Double ->  Point Double
g (Pt x y) = Pt (x+2) (y+2)
```

Таким образом, введённый тип `Point` является полиморфным.

Также возможно создание типов без каких-либо параметров:

```
data PointOnMonitor = Point2D Int Int Color

mycolor :: PointOnMonitor -> Color
mycolor (Point2D x y c) = c

mycoord :: PointOnMonitor -> Point Int
mycoord (Point2D x y c) = Pt x y
```

В одном объявлении возможно совмещения перечисления и создания структуры:

```
data Color2 = Red | Blue | RGB Int Int Int deriving Show
```

и пример использования:

```
toInt Red  = RGB 255 0 0
toInt Blue = RGB 0 0 255
toInt x    = x

toName (RGB 255 0 0) = Red
toName (RGB 0 0 255) = Blue
toName _ = error "Not defined!"
```

Отметим, что кортеж `(x,y)` и его конструкторы типа и данных соответствует правилам конструирования алгебраических типов:

```
data (,) a b  = (,) a b
```

Именованные поля (записи)

Однако, более удобно использование так называемых именованных полей:

```
data Point = Point {pointx, pointy :: Double}
  deriving Show
```

Тогда допустимо такое использование именованных полей:

```
absPoint p = sqrt ( (pointx p)^2 + (pointy p)^2 )
```

или

```
p2 =  Point {pointx = 1.0, pointy = 2.2 }
```

Таким образом, отпадает нужда во введении *геттеров* и *сеттеров* (вроде функций, подобных `getX` из предыдущего раздела). И появляется возможность создавать функции с именованными аргументами.

При сопоставлении с образцом будем писать:

```
absPoint' (Point { pointx = x, pointy =y } ) =
  sqrt (x^2 + y^2)
```

или возможно использование «старого стиля»:

```
absPoint'' (Point x y) = sqrt (x^2 + y^2)
```

Изоморфные типы `newtype`

Работу конструкторов данных можно рассматривать как «навешивание» нового тэга к известным типам, для указания возможности использования их в других смыслах.

Собственно при создании значения нового типа и происходит такое оборачивание. Оно приносит дополнительные расходы. Чтобы избежать их в простейших случаях, Haskell предлагает использовать декларацию **`newtype`**. Эта декларация также позволяет создавать новые типы данных как и **`data`**, однако работает только в ситуации, когда у нас есть только один конструктор значений и одно поле.

Рассмотрим пример, взятый из статьи Difference between data and newtype in Haskell на StackOverflow:

```
data Book = Book Int Int

newtype Book = Book (Int, Int)
```

(во втором случае мы можем использовать только одно поле)

Вот что мы видим в первом случае:

Вот что мы видим во втором случае:

Здесь картинка описывает, как после компиляции будет представлена эта структура данных. Нет лишней обёртки, тэг убран, и фактически, мы имеем тот же самый тип данных, что и «до оборачивания»!

Можно было бы сделать определение и таким образом:

```
data Book = Book (Int, Int)
```

Но накладные расходы все равно бы остались:

Кроме этого, **`newtype`** более строго проводит конструирование значений, не откладывая их «на потом».

Таким образом, **`newtype`** — это более эффективный аналог **`data`** для простых случаев. Кроме того, эта декларация позволяет создавать «изоморфные типы данных», которые

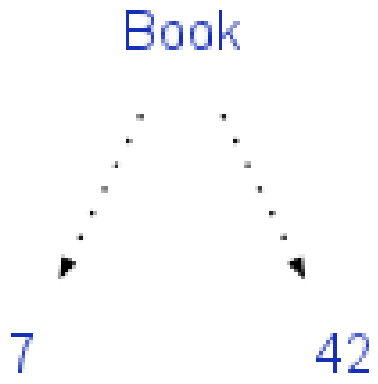

Рис. 1: Обёртка при использовании data

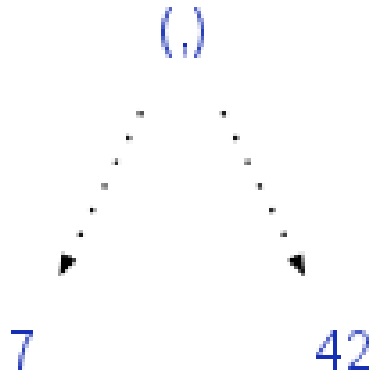

Рис. 2: Нет обёртки при использовании newtype

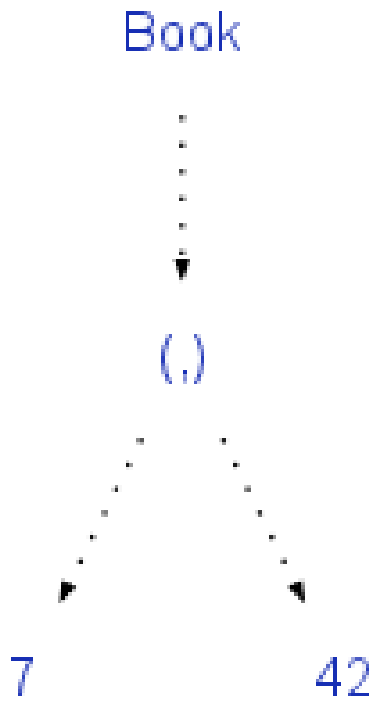

Рис. 3: Вновь обертка при использовании data

будут различаться компилятором, но будут иметь одинаковое внутреннее представление при исполнении.

Подведём итог, вот кратко, что можно сказать об этих декларациях:

data:

- может использовать несколько конструкторов значений
- конструкторы значений «ленивы» (если не дать спец. указаний)
- значения могут иметь несколько полей
- влияет и на компиляцию и на исполнение, во время исполнения несёт затраты на «обёртки»
- во время паттерн-матчинга вычисляет вглубь свое построение (по крайней мере частично, до «слабой нормально головной формы»)
- используется для создания новых типов данных

newtype:

- может иметь только один конструктор
- конструктор строгий
- значение может иметь только одно поле
- влияет только на компиляцию, во время исполнения нет лишних накладных расходов
- во время паттерн-матчинга вообще не вычисляется
- используется для создания структур более высокого уровня на основе существую-

щего типа с возможным добавлением новых операций или для удобства в различии с исходным типом

Рассмотрим пример.

```
newtype Email = MkEmail String
```

В данном случае новый тип `Email` позволяет просто ввести практически синоним для типа **`String`** в целях отличать его (в том числе и на уровне компилятора). Хотя при исполнении это не принесёт дополнительных расходов и будет обрабатываться как будто это строка.

Так как у нас только один конструктор, то использование именованного поля позволяет сразу же при объявлении ввести *десктруктор*:

```
newtype Email = MkEmail { adr :: String }
```

и тогда

```
mail1 = MkEmail "to Mr. Smith"

>adr mail1
"to Mr. Smith"
```

Кроме того, мы можем ввести операции, специфичные только для почтовых отправлений:

```
cost_of :: Email -> Int

cost_of (MkEmail _) = 10
```

Why is there “data” and “newtype” in Haskell?

Difference between **`data`** and **`newtype`** in Haskell

wiki.haskell: Newtype

Магия newtype в Haskell

Об изморфизме в Haskell

What does “isomorphic” mean (in Haskell)?

Isomorphism in newtype

youtube: Р.Душкин. Что такое изоморфные типы в языке Haskell?

youtube: Р.Душкин. Как определить экземпляры изоморфных типов?

:o(:o(:o(

Синонимы типов `type`

Если нам нужно ввести просто синоним какого-либо типа, то следует использовать декларацию **`type`**.

```
type String = [Char]
type Name = String
type Tochka = Point Float
```

type:

- создает альтернативное имя (синоним) типа
- никаких конструкторов
- никаких полей
- разбирается только во время компиляции, никаких накладных расходов во время исполнения
- никакого нового типа не создаётся
- во время паттерн-матчинга работает как оригинальный тип