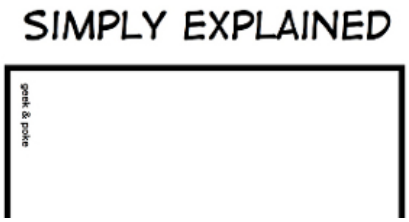
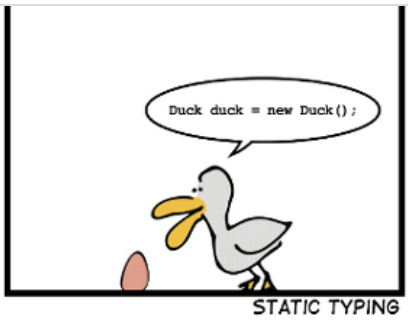
Fai 3 декабря 2012 в 20:46

Ликбез по типизации в языках программирования

Программирование

Из песочницы





Эта статья содержит необходимый минимум тех вещей, которые просто необходимо знать о типизации, чтобы не называть динамическую типизацию злом, Lisp — бестиповым языком, а C — языком со строгой типизацией.

В полной версии находится подробное описание всех видов типизации, приправленное примерами кода, ссылками на популярные языки программирования и показательными картинками.

Рекомендую прочитать сначала краткую версию статьи, а затем при наличии желания и полную.

Краткая версия

Языки программирования по типизации принято делить на два больших лагеря — *типизированные* и *нетипизированные* (*бестиповые*). К первому например относятся C, Python, Scala, PHP и Lua, а ко второму — язык ассемблера, Forth и Brainfuck.

Так как «бестиповая типизация» по своей сути — проста как пробка, дальше она ни на какие другие виды не делится. А вот типизированные языки разделяются еще на несколько пересекающихся категорий:

- *Статическая / динамическая* типизация. Статическая определяется тем, что конечные типы переменных и функций устанавливаются на этапе компиляции. Т.е. уже компилятор на 100% уверен, какой тип где находится. В динамической типизации все типы выясняются уже во время выполнения программы.

  Примеры:
  Статическая: C, Java, C#;
  Динамическая: Python, JavaScript, Ruby.

- *Сильная* / *слабая* типизация (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.

- *Явная* / *неявная* типизация. Явно-типизированные языки отличаются тем, что тип новых переменных / функций / их аргументов нужно задавать явно. Соответственно языки с неявной типизацией перекладывают эту задачу на компилятор / интерпретатор.

  Примеры:
  Явная: C++, D, C#
  Неявная: PHP, Lua, JavaScript

Также нужно заметить, что все эти категории пересекаются, например язык C имеет статическую слабую явную типизацию, а язык Python — динамическую сильную неявную.

Тем-не менее не бывает языков со статической и динамической типизаций одновременно. Хотя забегая вперед скажу, что тут я вру — они действительно существуют, но об этом позже.

Пойдем дальше.

Подробная версия

Если краткой версии Вам показалось недостаточно, хорошо. Не зря же я писал подробную? Главное, что в краткой версии просто невозможно было уместить всю полезную и интересную информацию, а подробная будет возможно слишком длинной, чтобы каждый смог ее прочесть, не напрягаясь.

Бестиповая типизация

В бестиповых языках программирования — все сущности считаются просто последовательностями бит, различной длины.

Бестиповая типизация обычно присуща низкоуровневым (язык ассемблера, Forth) и эзотерическим (Brainfuck, HQ9, Piet) языкам. Однако и у нее, наряду с недостатками, есть некоторые преимущества.

Преимущества

- Позволяет писать на предельно низком уровне, причем компилятор / интерпретатор не будет мешать какими-либо проверками типов. Вы вольны производить любые операции над любыми видами данных.
- Получаемый код обычно более эффективен.
- Прозрачность инструкций. При знании языка обычно нет сомнений, что из себя представляет тот или иной код.

Недостатки

- Сложность. Часто возникает необходимость в представлении комплексных значений, таких как списки, строки или структуры. С этим могут возникнуть неудобства.
- Отсутствие проверок. Любые бессмысленные действия, например вычитание указателя на массив из символа будут считаться совершенно нормальными, что чревато трудноуловимыми ошибками.

- Низкий уровень абстракции. Работа с любым сложным типом данных ничем не отличается от работы с числами, что конечно будет создавать много трудностей.

Сильная безтиповая типизация?

Да, такое существует. Например в языке ассемблера (для архитектуры x86/x86-64, других не знаю) нельзя ассемблировать программу, если вы попытаетесь загрузить в регистр cx (16 бит) данные из регистра rax (64 бита).

```
mov cx, eax ; ошибка времени ассемблирования
```

Так получается, что в ассемлере все-таки есть типизация? Я считаю, что этих проверок недостаточно. А Ваше мнение, конечно, зависит только от Вас.

Статическая и динамическая типизации

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

Некоторым людям может показаться, что статическая типизация слишком ограничена (на самом деле так и есть, но от этого давно избавились с помощью некоторых методик). Некоторым же, что динамически типизированные языки — это игра с огнем, но какие же черты их выделяют? Неужели оба вида имеют шансы на существование? Если нет, то почему много как статически, так и динамически типизированных языков?

Давайте разберемся.

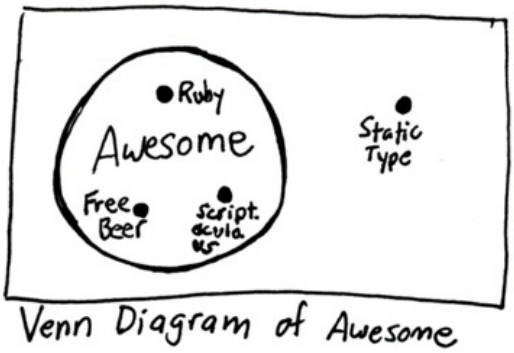
Преимущества статической типизации

- 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.



Обобщенное программирование

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

Как же мы будем ее решать? Решим ее на 3-ех разных языках: одном с динамической типизацией и двух со статической.

Алгоритм поиска я возьму один из простейших — 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)
```

Как видите, все просто и никаких проблем с тем, что список может содержать хоть числа, хоть списки, хоть другие массивы нет. Очень хорошо. Давайте пойдем дальше — решим эту-же задачу на Си!

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);
}
```

Ну, каждая функция в отдельности похожа на версию из Python, но почему их три? Неужели статическое программирование проиграло?

И да, и нет. Есть несколько методик программирования, одну из которых мы сейчас рассмотрим. Она называется обобщенное программирование и язык C++ ее неплохо поддерживает. Давайте посмотрим на новую версию:

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);
}
```

Хорошо! Это выглядит не сильно сложнее чем версия на Python и при этом не пришлось много писать. Вдобавок мы получили реализацию для всех массивов, а не только для 3-ех, необходимых для решения задачи!

Эта версия похоже именно то, что нужно — мы получаем одновременно плюсы статической типизации и некоторые плюсы динамической.

Здорово, что это вообще возможно, но может быть еще лучше. Во-первых обобщенное программирование может быть удобнее и красивее (например в языке Haskell). Во-вторых помимо обобщенного программирования также можно применить полиморфизм (результат будет хуже), перегрузку функций (аналогично) или макросы.

Статика в динамике

Также нужно упомянуть, что многие статические языки позволяют использовать динамическую типизацию, например:

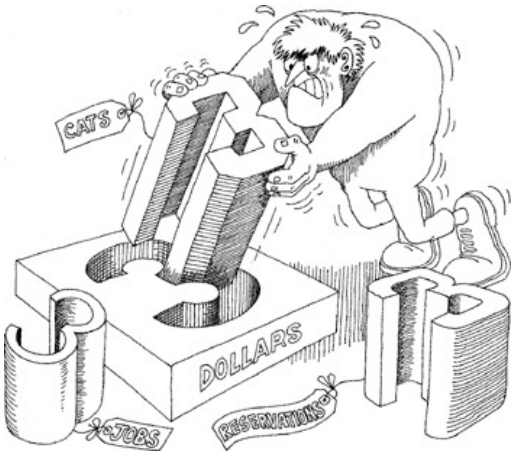
- C# поддерживает псевдо-тип dynamic.
- F# поддерживает синтаксический сахар в виде оператора ?, на базе чего может быть реализована имитация динамической типизации.
- Haskell — динамическая типизация обеспечивается модулем Data.Dynamic.
- Delphi — посредством специального типа Variant.

Также, некоторые динамически типизированные языки позволяют воспользоваться преимуществами статической типизации:

- Common Lisp — декларации типов.
- Perl — с версии 5.6, довольно ограниченно.

Итак, идем дальше?

Сильная и слабая типизации



Языки с сильной типизацией не позволяют смешивать сущности разных типов в выражениях и не выполняют никаких автоматических преобразований. Также их называют «языки с строгой типизацией». Английский термин для этого — strong typing.

Слабо типизированные языки, наоборот всячески способствуют, чтобы программист смешивал разные типы в одном выражении, причем компилятор сам приведет все к единому типу. Также их называют «языки с нестрогой типизацией». Английский термин для этого — weak typing.

Слабую типизацию часто путают с динамической, что совершенно неверно. Динамически типизированный язык может быть и слабо и сильно типизирован.

Однако мало, кто придает значение строгости типизации. Часто заявляют, что если язык статически типизирован, то Вы сможете отловить множество потенциальных ошибок при компиляции. Они Вам врут!

Язык при этом должен иметь еще и сильную типизацию. И правда, если компилятор вместо сообщения об ошибке будет просто прибавлять строку к числу, или что еще хуже, вычтет из одного массива другой, какой нам толк, что все «проверки» типов будут на этапе компиляции? Правильно — слабая статическая типизация еще хуже, чем сильная динамическая! (Ну, это мое мнение)

Так что-же у слабой типизации вообще нет плюсов? Возможно так выглядит, однако несмотря на то, что я ярый сторонник сильной типизации, должен согласиться, что у слабой тоже есть преимущества.

Хотите узнать какие?

Преимущества сильной типизации

- Надежность — Вы получите исключение или ошибку компиляции, взамен неправильного поведения.
- Скорость — вместо скрытых преобразований, которые могут быть довольно затратными, с сильной типизацией необходимо писать их явно, что заставляет программиста как минимум знать, что этот участок кода может быть медленным.
- Понимание работы программы — опять-же, вместо неявного приведения типов, программист пишет все сам, а значит примерно понимает, что сравнение строки и числа происходит не само-собой и не по-волшебству.
- Определенность — когда вы пишете преобразования вручную вы точно знаете, что вы преобразуете и во что. Также вы всегда будете понимать, что такие преобразования могут привести к потере точности и к неверным результатам.

Преимущества слабой типизации

- Удобство использования смешанных выражений (например из целых и вещественных чисел).

- Абстрагирование от типизации и сосредоточение на задаче.
- Краткость записи.

Ладно, мы разобрались, оказывается у слабой типизации тоже есть преимущества! А есть ли способы перенести плюсы слабой типизации в сильную?

Оказывается есть и даже два.

Неявное приведение типов, в однозначных ситуациях и без потерь данных

Ух… Довольно длинный пункт. Давайте я буду дальше сокращать его до «ограниченное неявное преобразование» Так что же значит однозначная ситуация и потери данных?

Однозначная ситуация, это преобразование или операция в которой сущность сразу понятна. Вот например сложение двух чисел — однозначная ситуация. А преобразование числа в массив — нет (возможно создастся массив из одного элемента, возможно массив, с такой длинной, заполненный элементами по-умолчанию, а возможно число преобразуется в строку, а затем в массив символов).

Потеря данных это еще проще. Если мы преобразуем вещественное число 3.5 в целое — мы потеряем часть данных (на самом деле эта операция еще и неоднозначная — как будет производиться округление? В большую сторону? В меньшую? Отбрасывание дробной части?).

Преобразования в неоднозначных ситуациях и преобразования с потерей данных — это очень, очень плохо. Ничего хуже этого в программировании нет.

Если вы мне не верите, изучите язык PL/I или даже просто поищите его спецификацию. В нем есть правила преобразования между ВСЕМИ типами данных! Это просто ад!

Ладно, давайте вспомним про ограниченное неявное преобразование. Есть ли такие языки? Да, например в Pascal Вы можете преобразовать целое число в вещественное, но не наоборот. Также похожие механизмы есть в C#, Groovy и Common Lisp.

Ладно, я говорил, что есть еще способ получить пару плюсов слабой типизации в сильном языке. И да, он есть и называется полиморфизм конструкторов.

Я поясню его на примере замечательного языка Haskell.

Полиморфные конструкторы появились в результате наблюдения, что чаще всего безопасные неявные преобразования нужны при использовании числовых литералов.

Например в выражении `pi + 1`, не хочется писать `pi + 1.0` или `pi + float(1)`. Хочется написать просто `pi + 1`!

И это сделано в Haskell, благодаря тому, что у литерала 1 нет конкретного типа. Это ни целое, ни вещественное, ни комплексное. Это же просто число!

В итоге при написании простой функции `sum x y`, перемножающей все числа от x до y (с инкрементом в 1), мы получаем сразу несколько версий — sum для целых, sum для вещественных, sum для рациональных, sum для комплексных чисел и даже sum для всех тех числовых типов что Вы сами определили.

Конечно спасает этот прием только при использовании смешанных выражений с числовыми литералами, а это лишь верхушка айсберга.

Таким образом можно сказать, что лучшим выходом будет балансирование на грани, между сильной и слабой типизацией. Но пока идеальный баланс не держит ни один язык, поэтому я больше склоняюсь к сильно типизированным языкам (таким как Haskell, Java, C#, Python), а не к слабо типизированным (таким как C, JavaScript, Lua, PHP).

Ладно, пойдем дальше?

Явная и неявная типизации

"Now! *That* should clear up a few things around here!"

Язык с явной типизацией предполагает, что программист должен указывать типы всех переменных и функций, которые объявляет. Английский термин для этого — explicit typing.

Язык с неявной типизацией, напротив, предлагает Вам забыть о типах и переложить задачу вывода типов на компилятор или интерпретатор. Англиски термин для этого — implicit typing.

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

Есть ли плюсы у каждого вида, и опять же, есть ли их комбинации и есть ли языки с поддержкой обоих методов?

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 входной аргумент, то в явно типизированном языке при изменении типа входного аргумента нужно будет изменить еще и тип временной переменной.

Хорошо, видно, что оба подхода имеют как плюсы так и минусы (а кто ожидал чего-го еще?), так давайте поищем способы комбинирования этих двух подходов!

Явная типизация по-выбору

Есть языки, с неявной типизацией по-умолчанию и возможностью указать тип значений при необходимости. Настоящий тип выражения транслятор выведет автоматически. Один из таких языков — Haskell, давайте я приведу простой пример, для наглядности:

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

-- Явное указание типа
add :: (Integer, Integer) -> Integer
add (x, y) = x + y
```

Примечание: я намерено использовал некаррированную функцию, а также намерено записал частную сигнатуру вместо более общей `add :: (Num a) => a -> a -> a`*, т.к. хотел показать идею, без объяснения синтаксиса Haskell'a.

* Спасибо @int_index за нахождение ошибки.

Хм. Как мы видим, это очень красиво и коротко. Запись функции занимает всего 18 символов на одной строчке, включая пробелы!

Однако автоматический вывод типов довольно сложная вещь, и даже в таком крутом языке как Haskell, он иногда не справляется. (как пример можно привести ограничение мономорфизма)

Есть ли языки с явной типизацией по-умолчанию и неявной по-необходимости? Кон
ечно.

Неявная типизация по-выбору

В новом стандарте языка C++, названном C++11 (ранее назывался C++0x), было введено ключевое слово auto, благодаря которому можно заставить компилятор вывести тип, исходя из контекста:

```
Давайте сравним:
// Ручное указание типа
unsigned int a = 5;
unsigned int b = a + 3;

// Автоматический вывод типа
unsigned int a = 5;
auto b = a + 3;
```

Неплохо. Но запись сократилась не сильно. Давайте посмотрим пример с итераторами (если не понимаете, не бойтесь, главное заметьте, что запись благодаря автоматическому выводу очень сильно сокращается):

```
// Ручное указание типа
std::vector<int> vec = randomVector( 30 );
for ( std::vector::const_iterator it = vec.cbegin(); ... ) {
    ...
}

// Автоматический вывод типа
auto vec = randomVector<int>( 30 );
for ( auto it = vec.cbegin(); ... ) {
    ...
}
```

Ух ты! Вот это сокращение. Ладно, но можно ли сделать что-нибудь в духе Haskell, где тип возвращаемого значения будет зависеть от типов аргументов?

И опять ответ да, благодаря ключевому слову decltype в комбинации с auto:

```
// Ручное указание типа
int divide( int x, int y ) {
    ...
}

// Автоматический вывод типа
auto divide( int x, int y ) -> decltype(x / y) {
    ...
}
```

Может показаться, что эта форма записи не сильно хороша, но в комбинации с обобщенным программированием (templates / generics) неявная типизация или автоматический вывод типов творят чудеса.

Некоторые языки программирования по данной классификации

Я приведу небольшой список из популярных языков и напишу как они подразделяются по каждой категории “типизаций”.

```
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
```

Примечания к таблице (за идею и напоминание о C# спасибо @qxfusion):

- C# — поддерживает динамическую типизацию, посредством специального псевдо-типа dynamic с версии 4.0. Поддерживает неявную типизацию с помощью dynamic и var.
- C++ — после стандарта C++11 получил поддержку неявной типизации с помощью ключевых слов auto и decltype. Поддерживает динамическую типизацию, при использовании библиотеки Boost (boost::any, boost::variant). Имеет черты как сильной так и слабой типизации.
- Common Lisp — стандарт предусматривает декларации типов, которые некоторые реализации могут использовать также для статической проверки типов.
- D — также поддерживает неявную типизацию.
- Delphi — поддерживает динамическую типизацию посредством специального типа Variant.

Возможно я где-то ошибся, особенно с CL, PHP и Obj-C, если по какому-то языку у Вас другое мнение — напишите в комментариях.

Заключение

Окей. Уже скоро будет светло и я чувствую, что про типизацию больше нечего сказать. Ой как? Тема бездонная? Очень много осталось недосказано? Прошу в комментарии, поделитесь полезной информацией.

И удачи!

Полезные ссылки

Прогопедия: типизации
Википедия: типизация
Квадранты типизации в языках программирования

Теги: программирование, типизация, языки программирования

Хабы: Программирование

Комментарии 177

Skerrigan 3 декабря 2012 в 21:02 +5

О как!.. В голове бывает все в кашку… теперь не будет благодаря вам.

→ **Fai** 3 декабря 2012 в 21:04 +71

Спасибо, был рад стараться. Если тема востребована, готов написать ликбез по парадигмам программирования в похожем стиле.

koldovsky 3 декабря 2012 в 21:12 +11

Востребована. Ждем.

d1pr3d 3 декабря 2012 в 21:25 +1

Буду рад прочитать.

injecto 3 декабря 2012 в 23:22 0

Если можно, то пишите глубже и хардкорнее :) Про типизацию ещё можно многое нарассказывать.
И спасибо за статью!

→ **Fai** 3 декабря 2012 в 23:53 +2

На здоровье! Нарассказывать действительно можно много: и про типизацию Хиндли-Милнера, и про автоматический вывод типов, и про безопасную типизацию (как раз что-то среднее между слабой и сильной), ну и конечно можно привести больше примеров кода — показать, где тот или иной вид типизации помогает писать код, а где — мешает.

Плюс, можно поподробнее описать как те или иные виды типизации реализованы различных языках программирования.

Однако статья итак вышла длинной, а парадигмы программирования — еще более глубокая тема.

В любом случае я хотел донести до читателя только те знания, которые с помогут в программировании, и что главное, будут интересны каждому. И в следующей статье я также попытаюсь извлечь из большой темы самую полезную и интересную информацию.

Zorkus 10 января 2013 в 01:43 0

habrahabr.ru/post/125250/ — про хиндли милнера уже написали

steck 4 декабря 2012 в 12:36 0

Как писалось ниже, типизация хорошо описана в книге Types and Programming Languages, Benjamin C. Pierce.
Книга переведена на русский язык. Можно нагуглить по названию «Типы в языках программирования».

injecto 4 декабря 2012 в 12:38 +1

Спасибо, она есть у меня :)

Vlad_IT 14 сентября 2015 в 14:24 0

Все еще ждем :-)

VolCh 28 сентября 2015 в 02:59 +2

Угу

// **anarleen** 3 декабря 2012 в 21:15 +9

Рекомендую почитать вот это еще по типизации.

blogs.msdn.com/b/ericlippert/archive/2012/10/15/is-c-a-strongly-typed-or-a-weakly-typed-language.aspx

Эрик Липперт интересно пишет про слабую и сильную типизацию C#.

> **What? You mean, is C# a strongly typed and a weakly typed language?**
>
> Yes, C# is a strongly typed language and a weakly typed language.

skywatcher 3 декабря 2012 в 21:19 0

Жаль, что вы Scala обошли стороной. К какому типу можно отнести ее?

Fai 3 декабря 2012 в 21:25 +2

Ух. Как раз сейчас учу этот дивный язык. В целом можно сказать, что типизация в нем статическая, строгая, неявная.

Хотя по-поводу явности/неявности сложно что-либо сказать, т.к. для var и val тип указывать не нужно (неявная с опциональной явностью), а для def нужно указывать типы аргументов (полностью явная).

senia 3 декабря 2012 в 23:37 0

Про то, что она сильная, тоже не все так однозначно. При желании можно наплодить неявных преобразований.
И не будем вспоминать про scala.Dynamic...

KilgortTraut 3 декабря 2012 в 23:46 0

Def от val отличается тем, что в val записывается сразу значение, а def определяется по нужде. Так в скала val x = square(2) сразу вычислится и будет равнятся 4 то def x = square(2) будет вычислятся каждый раз при вызове.
Так же в скала не обязательно указывать тип возвращаемого значения функцие, если она не рекурсивная.

В скале есть элементы как динамической типизации и слабой(оператор implicit позволяющий описывать неявные приведения типов)

Так же мне кажется, что данная методология все же ближе императивным языкам, в идеале функиональные языки по другому смотрят на эти вещи.

 **alexhemp** 4 декабря 2012 в 02:59 0

Динамической типизации в скале нет, есть вывод типов на этапе компиляции, позволяющий их иногда не указывать явно. Это «слабая» типизация в терминах автора статьи, но никак не динамическая :-)

 **Zay4egg** 4 декабря 2012 в 18:17 0

Ну есть еще lazy val, который вычислится по нужде один раз.

 **mstyura** 4 декабря 2012 в 00:39 0

Раз уж здесь зашла речь про скалу и вывод типов, то не могу не поделиться ссылкой на замечательный доклад Uncovering the Unknown: Principles of Type Inference

 **pfa** 4 декабря 2012 в 03:04 0

Собственно, явная она ровно настолько, чтобы быть строгой. В def'е KMK компилятор не может вывести типы аргументов, а потому они обязательны.

 **senia** 4 декабря 2012 в 08:35 0

Кстати, забавный вопрос: если учесть, что типизированные акторы почти не используются, а при написании кода с использованием акторов он может почти полностью состоять из отправки сообщений (вместо вызовов методов), то не привносит ли это в scala элемент динамической типизации?

Так что если покопаться:
Статическая/динамическая:

```
foo.method("a")
bar ! Message("a")
```

Сильная/слабая:

```
def test(b: Bar) = ???
implicit def fooToBar(f: Foo): Bar = ???
test(new Foo)
```

Явная по желанию:

```
val f1 = new Foo
val f2: Foo = getFoo
```

 **Alexey2005** 3 декабря 2012 в 21:23 +5

Языки с неявной типизацией тяготеют к write-only-коду. Писать на них легко, но вот чтение кода потом требует на порядок больше усилий.
Когда я после C++ попробовал JavaScript, меня просто дико выбешивало, что там совершенно невозможно понять, **что именно** надо передавать в функцию и что потом эта функция возвращает.
Сравним:

```
function getItemType(list,item)
```

или же

```
const std::string& getItemType(ProductsList &list,const int item)
```

Во первом случае надо шерстить код или документацию, а во втором можно сразу вызывать, лишь разок увидев всплывающую подсказку.

 **sebres** 3 декабря 2012 в 21:41 +3

а так?

```
// string getItemType (ProductsList | [Product] list, int item) --
// -- returns type of product with index "item" from list of products.
function getItemType(list,item)
```

Кстати, некоторые IDE иногда даже покажут вам этот коммент как ~~«всплывающую подсказку»~~ всплывающую инлайн доку.

 **eyeofhell** 3 декабря 2012 в 22:05 +3

> Кстати, некоторые IDE иногда даже покажут вам этот коммент как «всплывающую подсказку» всплывающую инлайн доку.

Что, увы, совершенно не помогает когда изучаем лог коммитов или читаем большие объемы кода — по наведению глаза на идентификатор IDE всплывающую подсказку не показывают, а наводть мышку на каждый интересный идентификатор спокойно увеличивает время ревью в несколько раз O_O.

 **Gorthauer87** 3 декабря 2012 в 22:08 +5

Имеет те же проблемы, что и венгерская нотация. Я бы вообще документирование кода ввел на уровне компиляторов и интерпретаторов, чтобы они при виде явного бреда в доке таки хотя бы варнинг кидали, а лучше ошибку.

 **maxatwork** 3 декабря 2012 в 21:45 +4

```
function getItemType(productsList, itemIndex) { ... }
```

… и все уже не так плохо. Хотя дисциплины нужно больше на порядок, да.

 **serf** 3 декабря 2012 в 21:45 0

По приведенному примеру помогли комментарии к методу, в стиле API дока.

>>> Языки с неявной типизацией тяготеют к write-only-коду.
Ага, некоторые просто напишут что нужно в виде отдельной функции, причем часто в глобальном пространстве и довольны, а ведь тут как нигде важна жесткая стандартизированная структура кода (конвенции, модульность, паттерны и тд), чтобы хоть как-то облегчить дальнейшую поддержку и развитие подобного «динамического» кода. Хотя бы модульность и классовый подход использовать желательно.

 **BlackFoks** 3 декабря 2012 в 22:01 +3

Дело привычки. Меня это так же бесило, когда я с C# на Ruby переходил. Потом привык и теперь, уверен, меня будет бесить необходимость указывать типы. :)

KilgortTraut 3 декабря 2012 в 23:50 0

Это дело личных предпочтений, мне вот приходится и на джаваскрипт пописывать и на питоне, так вот мне лично кажется, что лучше указывать типы — понятней код и меньше шанс накосячить передав не тот тип.

potan 3 ноября 2014 в 17:34 0

В языках Haskell и Scala в REPL можно прсмотреть выведенный компилятором тип. Помогает, когда разбираешься с чужей или забытой своей программой.

serf 3 декабря 2012 в 21:39 +2

Автоматический рефрактор (в том числе иерархия всех классов в одни-два клика, иерархия вызовов метода и тд) не является преимуществом статической типизации? Только не нужно утверждать что статический анализ кода позволит сделать тоже самое для динамического языка программирования (в JS можно всяких замыканий наворотить что анализатору будет не легко это разбирать).

>>> Легкость в освоении — языки с динамической типизацией обычно очень хороши для того, чтобы начать программировать.
А еще гики любят :)

PS сам пишу на Java и JavaScript.

serf 3 декабря 2012 в 21:47 0

«Автоматический рефакторинг» конечно имелся ввиду.

gleb_kudr 3 декабря 2012 в 21:49 +4

Троллинг мод он:

C#:
var foo — это какая типизация?
dynamic foo — а это какая?

;)

Fai 3 декабря 2012 в 21:50 0

Конечно правильное замечание, но я учел многие исключения из классического разделения.

«Также нужно упомянуть, что многие статические языки позволяют использовать динамическую типизацию, например:

C# поддерживает псевдо-тип dynamic.
...»

Gorthauer87 3 декабря 2012 в 22:10 0

Да и C++ имеет всякие типы вроде boost::any, boost::variant и QVariant, быть может когда нибудь такая штука и в стандарт попадет. В принципе эмулировать динамическую типизацию в ЯП со слабой статической типизацией достаточно просто до определенного предела.

Fai 3 декабря 2012 в 22:21 −2

Да и про ООП языки вроде Java и C# нельзя сказать, что они статически типизированы.

Например если функция имеет сигнатуру `A do_something( B b, C c );` то мы не можем быть уверены в том, что будут переданы именно экземпляры B и C, а возвращен экземпляр A, т.к. на деле там могут быть наследники этих классов или даже null.

Конечно в общем случае это статическая типизация, однако черты динамической так же присутствуют.

senia 4 декабря 2012 в 11:03 +2

Все-таки по общепринятой терминологии наследование — это отношение «является». Так что экзепляр наследника B является экземпляром B.

Ну а null не достаточно для того. чтобы говорить о динамичкеской типизации.

barker 4 декабря 2012 в 12:24 0

Ну и null на этапе компиляции тоже имеет соответствующий тип, очевидно.

senia 4 декабря 2012 в 12:47 +2

С null все сложно. Можно, конечно, говорить о том, что null является экземпляром класса со всеми методам, переопределенными так, что они бросают NRE.
Но ведь бывают не переопределяемые методы и поля, доступ к которым не может быть переопределен броском исключения.

Все-таки null — хак над системой типов. Кстати, как показывают некоторые языки (например Haskell и Scala), хак этот совершенно не обязателен и даже вреден.

Даже том же C# ввели Nullable, который имеет нечто общее с Maybe (и Option), хоть и не является полноценным аналогом, да и поздно в C# от null избавляться.

barker 4 декабря 2012 в 12:56 0

Ну, допустим, во многих случаях при связывании на этапе компиляции нереально понять какой метод вязать при неуказании конкретного типа для null. Пусть даже та же Java, например, как быть компилятору в случае двух методов: blabla(A a) и blabla(B b) и вызове: blabla(null)?

senia 4 декабря 2012 в 13:03 0

Допустим A и B — интерфейсы. Тогда если C реализует оба интерфейса, то мы получим ровно тот же вопрос в случае blabla(new C()).

В Java, как и в C# null — хак над системой типов.
В Scala же, например, хоть и не рекомендуется использовать null, но для совместимости с Java он есть и более того встроен в систему типов довольно забавным образом:
null является единственным экземпляром класса Null, который является наследником всех ссылочных типов (наследников AnyRef).
Таким образом ситуация blabla(null) и blabla(new C()) не просто вызывают один и тот же вопрос, как в Java, но и действительно являются одинаковыми.

barker 4 декабря 2012 в 13:11 0

В данном случае разумеется. Но ведь при этом выражение new C() обладает типом? :) Плюс и там и там этот вопрос можно решить явным указанием типа, не так ли? (A)new C() или (A)null.

senia 4 декабря 2012 в 13:38 0

Так я изначально утверждал, что ситуация двоякая:
С одной стороны null, якобы, является экземпляром B, а с другой нарушает не только принцип

подстановки, но и правила. распространяющиеся на все другие экземпляры (невозможность броска исключения при обращении к полю или не переопределяемому методу).

yorick_kiev_ua 4 декабря 2012 в 12:58 +1

Статическая.
Динамическая.

А в чём троллинг?

nsinreal 4 декабря 2012 в 13:22 0

var foo — статическая типизация с ограниченным автоматическим выводом типов
dynamic foo — статическая типизация с выключенной статической проверкой типов (так написано в msdn)

mwizard 3 декабря 2012 в 22:03 +1

И ни строчки о D :)

burjui 3 декабря 2012 в 23:59 +3

Видимо, автор решил взять для примера только зрелые языки. Всё-таки, как бы мы ни любили D, он до сих пор не устаканился, да и компилятор всё ещё решето.

mwizard 4 декабря 2012 в 03:02 +1

Увы, увы...

dmitriid 3 декабря 2012 в 22:27 0

Наконец-то есть статья, которой можно всем тыкать в спорах про типизацию :) Спасибо!

gribozavr 3 декабря 2012 в 22:38 +2

> нетипизированные (бестиповые).

Я не согласен с таким термином. Нельзя в термин вкладывать смысл что типов нет (как это вообще?) если на самом деле они есть (как вы сами сказали — битовые последовательности различной длины, можно давать и другие определения). И кроме того, если над ними ещё и производится проверки типов!

Смотрите сами: если вы оперируете какими-то значениями, то все такие допустимые значения принадлежат к какому-то множеству. И это множество и задаёт «тип». Поэтому я считаю сам термин «безтиповые» или «нетипизированные» некорректным.

Mrrl 3 декабря 2012 в 23:48 0

При необходимости можно и с C работать как с «нетипизированным» языком: все объекты захватывать с помощью malloc, описывать как char* (или int*) и завести там свою структуру «типов», подходящую конкретной задаче. И в функциях самостоятельно разбираться с семантикой каждого конкретного объекта. Базовые типы C останутся только для общения с процессором (как в ассемблере).

gribozavr 3 декабря 2012 в 23:52 0

И как это противоречит тому, что я сказал? Вы же будете использовать тип int? Будете. А ваша собственная структура «типов» будет только у программиста в голове.

Mrrl 4 декабря 2012 в 00:05 0

Тип int я буду использовать в той же мере, в какой использую регистр eax в ассемблере. Как я и сказал, для общения с процессором. И моя структура «типов» действительно будет только у программиста в голове, а никак не в языке — множество значений и операции над объектами существовать будут, но несколько выше, чем средства типизации, предоставляемые языком. В программе-то типы будут. Но это не те типы, что предусмотрены языком.

gribozavr 4 декабря 2012 в 00:16 +1

Во многих (я бы даже сказал «во всех», но для этого мне нужно проверить все) реальных программах система типов в какой-то мере в голове у программиста, потому что ни встроенные типы, ни средства создания типов не могут точно смоделировать предметную область. Вот например:

```
char c = get();
if (c >= '0' && c <= '9') {
  int digit = c - '0';
```

В большинстве языков вы никак не ограничите digit промежутком [0; 9] на уровне языка. И тип такой вам не дадут создать.

Mrrl 4 декабря 2012 в 00:25 0

Да, так и есть. Правда, боюсь, что через какое-то время за такой код будут бить ногами — скажут, что надо писать

```
Digit D(get());
if(D.IsValid()){
    int digit=D.Value();
```

— или как-нибудь еще более надёжно… Правда, это уже не Си. Но и на Си можно перевести, локализовав возможные некорректности в одном участке кода.

nwalker 4 декабря 2012 в 11:16 0

>> ограничить digit промежутком [0; 9] на уровне языка. И тип такой… создать.
Это же dependent types, да?

Mrrl 4 декабря 2012 в 11:30 +1

В каких-то языках это было. Не то в Паскале, не то в PL… давно было, не помню.

tyomitch 5 декабря 2012 в 00:40 +3

В Паскале было, называлось range types.
До сих пор скучаю по этой фиче в «мейнстримовых» языках — безмерно помогает самодокументируемости кода.

gribozavr 4 декабря 2012 в 12:18 0

Нет, диапазон же известен статически.

Fai 3 декабря 2012 в 23:57 0

Согласен, бестиповый язык — не совсем удачное определение для ассемблера или форта.

Однако это устоявшийся термин, плюс, я не знаю чем бы его можно было заменить.

gribozavr 4 декабря 2012 в 00:01 0

Понимаете, я не вижу даже где вы проводите черту. В тех же ассемблерах, например, из-за ограничений системы команд часто даже побитовое преобразование целого числа в число с плавающей точкой выполняется или отдельной командой или через память или ещё каким-то специальным образом. В данном отношении Си не менее «бестиповый», но тем не менее, вы его не относите к бестиповым.

Mrrl 4 декабря 2012 в 00:09 0

Может быть, отличие в том, что в Си целые и вещественные числа различаются, когда они находятся в памяти (если не пользоваться преобразованиями типов указателей), а в ассемблере — только когда они попали в регистры?

gribozavr 4 декабря 2012 в 00:13 0

> если не пользоваться преобразованиями типов указателей

Ничего себе «если»! В Си вы даже malloc() без невного преобразования типа указателя вызвать не сможете.

> а в ассемблере — только когда они попали в регистры?

А в большинстве ассемблеров нет типа «указатель» (в некоторых есть, там где для указателей специальные регистры). Указатель — число нужного размера.

Mrrl 4 декабря 2012 в 00:33 +1

Вызвать malloc я смогу. Вот воспользоваться его результатом — не очень. Оно и понятно, в памяти, захваченной malloc'ом пока еще нет объектов какого-то определенного типа. Её надо сначала разметить (хотя бы преобразовав указатель к нужному типу). Тогда она перейдет из «никакого» типа в определенный...

Про тип «указатель» в ассемблере я ничего не говорил. Но в 8086 они были и занимали целых два регистра (es:bx и т.п.) И аксиоматика чисел на них не очень работала.

matiouchkine 4 декабря 2012 в 01:28 +2

И воспользоваться результатом запросто. Вы туда сможете `memcpy` делать как душе заблагорассудится. Я прототип некой файловой системы так писал. Не будет работать только синтаксический сахар типа ++ и прочего люрекса.

merlin-vrn 4 декабря 2012 в 11:58 +2

Что важно, memcpy не волнует, какие там типы. Копирует и всё.

matiouchkine 4 декабря 2012 в 14:16 0

Пару лет назад была смешная история. Кто-то нашел более быстрый алгоритм для `memcpy`, который тут же закоммитили в федорины glibc.
Оп-па — поломалось что-то совершенно на первый взгляд левое (воспроизведение звука, например).
Разбор полетов показал, что ядерщики пользовались копированием памяти с перекрытием региона (кажется, потому что быстрее, чем `memmove`). А новый алгоритм копировал память по-арабски (справа налево).

merlin-vrn 4 декабря 2012 в 14:44 +2

:) вы невнимательно прочитали пост Линуса

Ядерщики вообще тут не при чём. Функция memcpy — в glibc, а не в ядре.

Накосячили по факту вообще в адоби. Они в флешплеере для линукса копировали перекрывающие регионы (как и показал valgrind), а в документации указано, что так делать не надо и результат не определён.

Баг проявился, когда glibc, желая всё жосско приоптимизировать для какого-то очередного пентиума, начали копировать память задом наперёд. Вот тут-то неопределённость поведения и проявилась.

Линус там ниже распинается в духе, что «в таком случае, нужно было забить на оптимизацию, раз это ломает уже использующиеся популярные приложения».

matiouchkine 4 декабря 2012 в 16:54 0

Ой, простите. Тут был НЛО.

Mrrl 4 декабря 2012 в 07:43 +2

Между прочим, в C есть совершенно легальное средство работать с одной и той же памятью и как с int и как с double без преобразования указателей. Называется **union**. Правда, в коде он практически не встречается (или мне не повезло его встретить), но компиляторы его до сих пор поддерживают.
Так что C еще на шаг ближе к ассемблеру в смысле типизированности…

gribozavr 4 декабря 2012 в 12:20 0

> Между прочим, в C есть совершенно легальное средство работать с одной и той же памятью и как с int и как с double без преобразования указателей. Называется union. [...] но компиляторы его до сих пор поддерживают.

Потому что это единственный разрешённый стандартом способ это делать. А вот преобразования типов указателей во многих случаях — нарушение strict aliasing.

tyomitch 5 декабря 2012 в 00:46 +3

Регулярно вижу union в коде, работающем с форматами файлов, потому что там это естественное и прозрачное представление «неоднозначного» формата.

Но не только там. В самый недавний раз видел его в нутрях Zend Engine, например.

grossws 27 августа 2016 в 02:57 +1

Ещё классика — в сетевых протоколах.

deneb42 4 декабря 2012 в 20:16 +1

Поддержу, типы есть всегда. Просто иногда у языка есть, например, ровно один тип значений. Пример: MUMPS/M, там всё — строка.
А ассемблер скорее вполне себе строго типизированный, просто типы — это числа разной длины и всё, других типов нет.

VladVR 4 декабря 2012 в 23:42 −1

Ну да, возможно правильнее сказать однотиповый, нежели безтиповый.
А невозможность скопировать весемь байт туда где могут лежать только 4 — это не строгая типизация, это просто здравый смысл. Вместо этого есть опкод, который копирует младшие байты. Ему в теории можно было бы назначить алиас, если так хочется типового абсолютизма. Но это никому не нужно по понятным причинам.

VladVR 4 декабря 2012 в 23:24 0

В ассемблере нет проверки типов. Как и собственно типов. Есть фиксированный набор команд, который попросту включает в себя не все варианты. Все что делает ассемблер — ищет по таблице. Если не может скопировать из одного регистра в другой, то это значит у процессора нет такого опкода, а не то, что где то там лексическая проверка выдала запрет.
Ну либо я сильно отстал от ассемблеров, раньше было так.

gribozavr 4 декабря 2012 в 23:50 0

> Все что делает ассемблер — ищет по таблице.

Вы тоже не можете провести точно границу. Разве граница в том, что «безтиповые» ищут по таблице?

Лексическая/синтаксическая/семантическая проверка в компиляторе ЯВУ тоже может быть реализована таблично. А в ассемблере может быть реализована и нетаблично, а алгоритмически.

VladVR 5 декабря 2012 в 00:09 0

Граница в том, что mov ax, bx и mov bx, ax это две разные команды, а не одна команда с двумя аргументами, куда что-то можно подставить, а что-то нет.
Тут алгоритмически можно что-то сделать только если там есть явная зависимость, ее может и не быть. И даже если она есть, то тут первичны команды, а алгоритм под них подстраивается, а в строгой типизации наоборот, первичен алгоритм или набор правил, а команды по этим правилам формируются.

gribozavr 5 декабря 2012 в 00:17 +1

> Граница в том, что mov ax, bx и mov bx, ax это две разные команды, а не одна команда с двумя аргументами, куда что-то можно подставить, а что-то нет.

Вы не поверите, но это одна и та же команда с двумя полями, куда подставляется любой номер регистра. (Посмотрите в интеловском мануале: mov r/m16, r16.)

Mrrl 5 декабря 2012 в 00:24 +1

При этом mov ax,bx и mov bx,es — остаются разными командами… А написать mov es,ds вообще нельзя.

gribozavr 5 декабря 2012 в 00:39 +1

Потому что это mov r/m16, Sreg. А mov Sreg, Sreg вообще нет. Видите, даже тут выходит что-то похожее на типы.

Mrrl 5 декабря 2012 в 00:42 0

Я-то вижу. Мне dword ptr хватает, чтобы увидеть, что базовые типы есть. А вот с пользовательскими — полная свобода, делай что хочешь, лишь бы не побили.

VladVR 5 декабря 2012 в 21:13 −1

Типы указателей не считаются. Речь о типах данных.

Mrrl 5 декабря 2012 в 21:25 0

Ага.
mov byte ptr es:[bx],1
и

mov word ptr es:[bx],1
— разные действия. (в синтаксисе могу ошибаться, но по-моему, обе команды существуют).

VladVR 5 декабря 2012 в 21:35 0

Именно, что разные действия. Вопрос кто запретит их оба выполнить над одним и тем же адресом, кто обеспечит данным этот самый «тип».

Mrrl 5 декабря 2012 в 22:35 0

А чем они отличаются от
(char)es_bx=1;
и
(short)es_bx=1;
соответственно? По-моему, ничем. И можно задавать те же самые вопросы. Особенно если «типы указателей не считаются».

Другое дело, что пару команд
mov es:[bx],al
и
mov es:[bx],ax
мы так просто не адаптируем, у сишного указателя es_bx какой-то тип есть — и он определяет, что лежит в этой ячейке «по умолчанию», а у ассемблерного es:[bx] типа нет — он «по умолчанию» адаптируется к типу второго арумента (а при неоднозначности — ругается).
Наверное, в этом можно увидеть разницу…

VladVR 6 декабря 2012 в 09:28 0

Если пишешь на Си
char a;
short b;
А потом пишешь
a =1;
b = 1;
то компилятор, проверив тип данных находящихся в ячейке сам назначит нужную ассемблерную команду. Если же пишешь на асме сам, то никто за тебя не проверит тип ячейки и не запретит использовать «не ту» команду.
И в случае a =b; язык строгой типизации запретит, язык со слабой типизацией подберет сам нужную команду, а на асме сам программист должен будет решить что использовать, потому что, опять же, никто не запретит ему использовать al в той же команде mov, да и вобщем то хоть бы и ah, если программист сам для себя придумал конверсию.
В обоих случаях типизация данных(а не команд) лежит выше ассемблера, в компиляторе, либо в голове(блокнотике) программиста.

VladVR 5 декабря 2012 в 21:32 0

С точки зрения процессора это все равно разные команды. Говоря языком Си, это mov_ax_bx(); а не mov(ax,bx);

Собственно что определяет типизацию вообще — проверка типов. Ничто же не помешает скопировать int в uint, да хоть бы и в short. В любом из случаев это после компиляции всего лишь адрес в памяти. Но если проверка типа не прошла — исполнение будет запрещено с требованием предоставить явное преобразование.
Слабая типизация характеризуется ровно тем же, Проверкой типов. Только при невыполнении условия совпадения типов, будет автоматически подставлено неявное преобразование.

В ассемблере же проверки типов нет. Нигде, ни на одном из этапов. Тип всегда один, отличается лишь операция. Из одного и того же адреса памяти 16битный mov возмет 2 байта, а 32хбитный 4. И никто никогда не узнает, что там «однобайтовый тип» лежал. Пока код не упадет.

На примере арифметических операций, кстати еще лучше видно, что нет никаких типов, это даже было, помнится, в презентации amd — несколько каскадированных 32хбитных операций теперь схлопываются в одну, получаем выигрыш в

скорости. Любую операция, хоть бы и таке же 64хбитное сложение можно было сделать и на 8ми битном процессоре. Но теперь, когда АМД внедрила новый, никем ранее не использованный 8 байтовый тип данных… хехе.

зЫ все сдаюсь, пусть будет типизированный, бг

VladVR 3 декабря 2012 в 22:56 0

А можно пояснить, почему у C++ полусильная типизация? И относится ли сильная/слабая типизация к ООП в том числе или только к базовым типам?

barmaley_exe 3 декабря 2012 в 23:08 0

В C++ есть не-explicit конструкторы, операторы приведения типа, а ещё он сам с удовольствием принимает за bool ~~что угодно~~ int'ы и double, не равные нулю.

Fai 4 декабря 2012 в 00:02 0

В чистом Си — слабая типизация, в C++ же была взята его база и были изменены некоторые правила, из-за чего появились как черты как сильной так и слабой типизации.

nikitadanilov 3 декабря 2012 в 23:32 +4

> Terms like “dynamically typed” are arguably misnomers and should probably be replaced by “dynamically checked”.

Types and Programming Languages, Benjamin C. Pierce.

Beyondtheclouds 4 декабря 2012 в 00:29 0

Не по мотивам ШРИ ли ?:)

naryl 4 декабря 2012 в 01:19 +2

У вас очень ограниченный набор бестиповых языков. :) Всё какие-то близкие к железу.

Вот, Tcl тоже бестиповый с точки зрения программиста и был фактически бестиповым в реализации давным давно, пока его не начали оптимизировать. Только, в отличие от ваших примеров это высокоуровневый скриптовый язык и единственный тип в нём — строка. Даже вместо указателей на непрозрачные объекты (поток, сокет и т.д.) используется их уникальное имя (file1, sock42 и т.д.) т.е. строка.

Да, с CL всё правильно. Типизация сильная, динамическая, с опциональной явной. Вдобавок, некоторые реализации вроде SBCL при наличии деклараций типов пытаются его статически анализировать и выдавать ошибки или генерировать более оптимальный машинный код, но в первую очередь тип — это документация. Другие применения в стандарте не прописаны.

qxfusion 4 декабря 2012 в 17:51 +1

Да сколько уже можно — в Tcl ЕСТЬ И ДРУГИЕ ТИПЫ КРОМЕ STRING — посмотрите сорцы интерпретатора tclAssembly.c tclCmdIL.c

naryl 5 декабря 2012 в 12:53 0

Да, есть. Я прекрасно об этом знаю, т.к. писал для него пару новых типов. Но при разработке на Tcl о типах можно смело забыть, пока не окажется, что ваш продукт тормозит чуть больше чем хотелось бы.

Wyrd 4 декабря 2012 в 01:33 +1

Уберите, пожалуйста, сравнения вещественных чисел в стиле a == b. Нельзя их так сравнивать...

Mrrl 4 декабря 2012 в 07:49 +2

Почему нельзя? Конечно, если происхождение значений независимо, они, с большой вероятностью, не совпадут. Но если я положил в a какое-то большое число, а потом решил проверить, осталось ли оно там, или его заменили на меньшее — почему мне нельзя проверить равенство? Или если число лежит на отрезке [p,q], где p и q целые, и надо отдельно отработать случай a==q (при этом любое a=q-eps в этот случай не попадает). Почему я должен всех запутывать и писать a>=q?

Wyrd 4 декабря 2012 в 18:14 +2

> Но если я положил в a какое-то большое число, а потом решил проверить, осталось ли оно там, или его заменили на
> меньшее — почему мне нельзя проверить равенство?
А если его «заменили» на такое же, но посчитанное другим способом?
Да и вообще, я бы не дал «палец на отсечение», что равенство выполнится даже если никто ничего больше не писал. Есть, например, настройки оптимизации вещественных вычислений у компилятора, которые, кстати, умеют делать «unsafe» оптимизации.

> Или если число лежит на отрезке [p,q], где p и q целые, и надо отдельно отработать случай a==q
> (при этом любое a=q-eps в этот случай не попадает).
Если у вас есть такая логика в программе с вещественными числами, то 99%, что проблема кроется в архитектуре.

> Почему я должен всех запутывать и писать a>=q
А вот > вместо >= писать как раз можно. Если вы конечно уверены, что у вас там строго больше.

Вообще, по поводу «запутывания», на знаю как вас, но меня код типа IsWithinAccuracy(a, b) совсем не запутывает.
А вот a == b в случае вещественных числе запутывает и очень даже сильно. Потому что я автоматически пытаюсь понять, что имел в виду автор, если он сравнивает «напрямую». Должны быть очень веские причины писать такое сравнение. Я сходу даже придумать не могу такой причины.

Mrrl 4 декабря 2012 в 19:27 0

Задача, о которой идет речь — представление функции кусочно-линейной интерполяцией. Функция задана на отрезке от 0 до N, гарантируется, что аргумент — вещественное число — число лежит в этом диапазоне, при этом шансы, что оно равно N, сильно ненулевые. Если значение x==N, я возвращаю последний элемент массива, в остальных случаях беру n=(int)x; y=x-n и возращаю A[n]*(1-y)+A[n+1]*y. Где ошибка в архитектуре?

Wyrd 5 декабря 2012 в 13:37 0

Вместо вектора A должен быть hash_map<double, double>
Где key — координата x, value — значение функции.
Тогда вам не придется делать преобразование (int)x, потому что вместо этого будет что-то вроде A.lower_bound(x).
А нету преобразования — нету проблемы :)

Ну а в вашей реализации я бы внутри функции не закладывался на то, что выполняется условие x <= N, выполняться должно только x <= N + epsilon (а в этом случае нужно обязательно ставить >= вместо ==). Потому как совершенно неизвестно, как был получен x.

Mrrl 5 декабря 2012 в 13:55 0

Известно, что x — средняя G-компонента пикселей на фрагменте картинки. Так что она наверняка 0<=x<=255, причём значение 255 достигается, а больше — никогда. Но для безопасности действительно лучше написать x>=N, согласен. (а еще лучше — n>=N).
Про hash_map я не понял. Он быстрее вектора? Или занимает меньше памяти? Или автоматически выполняет интерполяцию? В чём его выигрыш?

Wyrd 5 декабря 2012 в 14:15 0

Оно не быстрее вектора, отличается по скорости на константу, и предназначено для хранения пар ключ-значение, быстрого добавления новых и быстрого (O(1)) поиска пары по ключу. Плюс его в том, что убивается связь индекс == (int)x. Индекс в хеш мапе можно сделать даблом, что позволяет хранить функции не только на промежутке 0 до N, а вообще на любом, да еще и бить их на разные куски (у вас между «реперными» точками всегда расстояние в 1.0). Памяти жрет раза в 2 больше (нужно ключ хранить). В общем выигрыш архитектурный в первую очередь. Ну, и еще потенциальная расширяемость.

> Известно, что x — средняя G-компонента пикселей на фрагменте картинки. Так что она наверняка 0<=x<=255, причём значение 255 достигается, а больше — никогда.
Не наверняка. Может получиться 255.00000000000001 если у вас пиксели все по 255

Mrrl 5 декабря 2012 в 14:37 0

Интересно, как вы вообще собираетесь использовать double в качестве ключа hash-map, если нет точного сравнения? Неужели hash-функция у вас работает с точностью до eps? А иначе вы значения никогда не найдете, потому что ключ будет каждый раз слегка другой.
А чтобы искать ближайшую или ближайшую левую точку, понадобится уже что-то вроде дерева. С поиском за O(log(N)).
Насчет 255.0000000000001 — интересно. Как это может быть? Пикселей на то, чтобы переполнить 52 бита мантиссы, у нас нет (потребовалась бы 16-терапиксельная картинка), поэтому сумма будет вычислена точно. Как деление целого числа на целое может дать неверный результат? Вы хотите сказать, что 9.0/3.0 это не обязательно 3.0?

Wyrd 5 декабря 2012 в 14:53 −1

насчет hash_map: оно умеет искать не только точное соотвествие, но и наиболее близкие элементы сверху/снизу, так что тут проблемы нету.

по поводу 255. Да согласен, не получается в данном случае, но я б все равно перестраховался. В конце-концов 0;255 может резко измениться на какой-нибудь float[0;1] в другом формате

Mrrl 5 декабря 2012 в 14:54 0

Сколько времени занимает поиск близких элементов? Тоже O(1)?

nsinreal 5 декабря 2012 в 15:54 +2

> насчет hash_map: оно умеет искать не только точное соотвествие, но и наиболее близкие элементы сверху/снизу, так что тут проблемы нету.

А как? Насколько я понимаю, оно работает на хэш-таблицах, а как эту фичу приделать туда я придумать не могу

Кстати, относительно проблемы @Mrrl — по мне так хорошее решение, но оно обломается если у него между ближайшими точками будет разное расстояние. Впрочем, я думаю это ему не грозит.

Mrrl 5 декабря 2012 в 08:12 0

И еще очень любопытно, как написать функцию сравнения вещественных чисел для сортировки. Если под равными числами понимать числа, равные с заданной точностью, то нарушится аксиоматика порядка (будет возможна ситуация a=b, b=c, a<c), и результат станет непредсказуемым.

barmaley_exe 5 декабря 2012 в 10:38 +3

Так для сортировки достаточно отношения строгого порядка, т.е. оператора < (std::sort требует именного строгого порядка, иначе может случиться всякое). Ну а если получится. что $a \leq b \leq c$, хотя в настоящей математике получилось бы $b < a = c$, то,

видимо, b столь ничтожно отличается от a и c, что нужно использовать более точные вычисления (какой-нибудь класс рациональных чисел, например), если мы хотим их различать.

Mrrl 5 декабря 2012 в 11:01 0

Ну, а если у нас массив A[n]=-n*eps/2, а сортировка вздумает проверить, надо ли его сортировать вообще? Условие A[n] <A[n-1] (при сравнении с точностью до eps) не выполняется ни разу (соседние элементы в этом массиве «равны»), поэтому функция имеет право ничего не делать. А различие между первым и последним элементом может быть совсем не ничтожным, причем не в ту сторону...

barmaley_exe 5 декабря 2012 в 11:06 +2

Если Вы работаете на пределе точности какого-нибудь double и используете тот самый double, то Вы делаете что-то не так.

Mrrl 5 декабря 2012 в 11:28 0

Кто говорит про предел точности? И вообще, как именно вы предлагаете проверять равенство вещественных чисел? Откуда брать порог точности?

Wyrd 5 декабря 2012 в 14:00 0

Есть 2 пути:
1 — взять заведомо малое значение, но достаточное, чтобы ошибка его превысила. Например, 1e-10. Это зависит от задачи. 1e-10 — это такой середнячок, и не очень много, и не очень мало, приблизительно середина точности дабл (точность дабл что-то около 17 знаков, точно не помню).
2 — «правильный», но сложный и зачастую не нужный: считать ошибку паралелльно со всеми расчетами. Для этого есть специальные формулы.

Подробнее — ищите мат часть в интернете

Wyrd 5 декабря 2012 в 13:55 0

Сравнивать с точностью до eps, обязательно только в том случае, если вы пытаетесь сравнить на РАВЕНСТВО. Любые неравенства сравнивуются «по ситуации». Т.е. если вам сравнение нужно, чтобы, скажем, отфильтровать неверные данные (например, меньше 0) и вы не уверены в точности параметра (он вычислен, скажем, а не из базы взят), то надо писать epsilon и округлять до нуля если вдруг все плохо. В сравнениях же, где важен порядок (сортировка) нужно обязательно сравнивать без eps.

С другой стороны, правильно написанное сравнение A[n]<A[n-1] с учетом eps, таки сортирует массив, но числа в пределах eps остаются при этом перемешанными.

ltwood 21 марта 2014 в 18:18 0

Еще раз повторю, что очень неправильно давать абсолютно некорректные рекомендации.

Сравнивать с точностью до фиксированного числа eps можно только в том случае, когда все сравниваемые числа имеют примерно одинаковый порядок.

С нулем всегда нужно сравнивать только на точное равенство и неравенство, иначе произойдет существенное сужение динамического диапазона используемых чисел.

Сравнение с учетом точности используется только при сравнении двух чисел, ни одно из которых не обязательно в точности равно нулю. При этом выбор величины параметра eps должен учитывать порядок сравниваемых чисел. Один из вариантов такой:

```
real abseps = (eps + 2.0*REAL_EPS) * (1.0 + max(fabs(a), fabs(b)));
```

Здесь eps — желаемая точность сравнения, REAL_EPS — абсолютная точность используемого вещественного типа real, abseps — та абсолютная точность, с которой необходимо сравнивать числа a и b.

ltwood 21 марта 2014 в 18:04 +1

«Флоатофобия» — не менее тяжелая болезнь, чем ~~студенческое~~ пофигистическое отношение к вещественным числам. Прежде чем всех пугать привидениями, пожалуйста, возьмите на себя труд познакомиться с предметом.

> Если у вас есть такая логика в программе с вещественными числами, то 99%, что проблема кроется в архитектуре.

Стандарт IEEE 754 гарантирует, что для целых чисел, сохраненных в виде вещественного числа все операции, не выводящие за пределы можества представимых целых чисел (в том числе и сравнение), выполняются точно и в полном соответствии с аксиомами целых чисел. Поэтому целые вещественные числа можно и нужно сравнивать на точное равенство и в некоторых языках присутствуют только вещественные числа.

barmaley_exe 22 марта 2014 в 13:36 0

> Стандарт IEEE 754 гарантирует, что для целых чисел, сохраненных в виде вещественного числа

Каких именно целых чисел? 10^300 тоже целое число, но представить точно 10^300+1 в 64 битах не получится.

Более того, в JS `Math.pow(2,63) == Math.pow(2,63)-1` (как и в Питоне `(2.0**63+1) == (2.0**63)`), т.е. точности double не хватит даже для int64 (что, разумеется, очевидно из того, что их битовая ёмкость одинакова).

ltwood 22 марта 2014 в 17:39 0

Очевидно, что в каждом случае имеются в виду (точно) представимые целые числа и этот диапазон отличается от диапазона представимых вещественных чисел.

VolCh 23 марта 2014 в 23:36 0

Основная проблема, по-моему, в том, что в мэйнстрим языках нет поддержки «из коробки» чисел с фиксированной точностью — целые или плавающие (как вариант только плавающие), — в то время как в реальной жизни они сплошь и рядом. И мало кто додумывается использовать плавающие числа при измерении, скажем, денег для копеек, а не рублей, при массы — граммов (или миллиграммов), а не килограммов (или граммов), что даст плюсы и целочисленной арифметики — точное сравнение конечных результатов, — и плавающей — более точное вычисление промежуточных результатов, большая гибкость при округлении.

ltwood 24 марта 2014 в 18:40 0

Не знаю, мне кажется, что это несколько иной круг задач, хотя они и довольно широко распространены. А вот исходники, в которых решают квадратное уравнение и ограничивают знаменатель константным eps, просто уже утомляют.

VolCh 25 марта 2014 в 01:22 0

Каждый о своём больном месте :(

Seter17 4 декабря 2012 в 03:15 +2

отличная для книга для тех кто хочет знать больше «Types and Programming Languages» Benjamin C. Pierce

НЛО прилетело и опубликовало эту надпись здесь

Mithgol 4 декабря 2012 в 11:15 +1

People familiar with type theory могли бы заметить, что все три языка, наиболее популярные в сайтостроении (JavaScript, PHP, а до PHP — Perl), имеют динамическую слабую неявную типизацию.

Что-то в этом есть, не правда ли?

steck 4 декабря 2012 в 11:57 +5

Конечно. Думаю, это произошло из-за постепенной эволюции языков.
Вначале был нужен шаблонизатор, который в подготовленную страничку вставит полученные из хранилища данные. Для таких задач не нужна ни статическая ни явная типизация, она только усложнит написания.

Задачи в браузере тоже не были глобальными. Джаваскрипт использовался маленькими функциями-вставками не превращая страницу в полноценное приложение. Простота и краткость были очень важными критериями.

Наследственность очень тяжело побороть.
В последнее время, я стал всё чаще видеть попытки использовать статические языки (ocaml, haskell, scala) для геренерации джаваскрипт кода.
Постепенно набирают популярность Asp.Net экосистема, Go, ocaml со стороны сервера. Есть попытки сделать что-то приближенное к современному понимаю типизации, например yesod для haskell.

Процесс медленный, но в задачах, сложность которых заставляет испытывать дикий восторг от возможности автоматической проверки хоть чего-либо, языки с сильной статической типизацией обгоняют всё остальное. Сложность задач решаемых при сайтостроении плавно приближается к такому уровню.

KvanTTT 5 декабря 2012 в 23:48 0

> В последнее время, я стал всё чаще видеть попытки использовать статические языки (ocaml, haskell, scala) для геренерации джаваскрипт кода.

Еще C# (Script#).

nsinreal 4 декабря 2012 в 13:09 +5

Предлагаю вам прочитать прекрасную статью: PHP: фрактал плохого дизайна. Людям нравится говно. Что-то в этом есть, не правда ли?

Ну да ладно, это все равно не имеет отношения к типизации, @steck уже объяснил чем была выгодна динамическая типизация в вебе.

kefirr 4 декабря 2012 в 14:34 +3

Ключевое слово — «популярные»

НЛО прилетело и опубликовало эту надпись здесь

VolCh 4 декабря 2012 в 07:03 +2

Как-то мне казалось, что есть типизация строгая (strict), а есть сильная (strong). Отличие, кажется: при строгой типизации вообще нет неявных преобразований типов, а сильная допускает безопасные преобразования. Но в любом случае strict сильнее чем strong, очень сильная типизация другими словами.

В PHP есть элементы опциональной явной сильной типизации — type hinting. Весьма ограниченная — только аргументы функций/методов, только объектные типы или массивы, — но есть. Например, `function has_permissions(User $user, array $permissions) {}` вызовет ошибку времени исполнения при попытке вызвать её с первым аргументом не класса User (или не имплементацией интерфейса User) или их наследников, а с вторым — не массивом. Но для скаляров это не реализовано, насколько я понимаю, по причине того, что PHP всё же слаботипизированный язык и type hinting приведёт к тому, что он практически не будет выполнять своей роли (передача строки вместо целого будет преобразовывать строку в целое) или нарушит единообразие, что неявные преобразования перестанут работать в некоторых случаях, а в PHP его и так мало :).

 **alan008** 4 декабря 2012 в 09:15 0

Delphi: статическая, сильная, явная. Добавьте, плиз, в сводную табличку :)

 **merlin-vrn** 4 декабря 2012 в 10:37 +3

странно, что ничего нет про duck typing — было бы уместно в этой статье

 **tyomitch** 5 декабря 2012 в 00:52 0

Про неё есть КДПВ :)

 **lolmaus** 24 октября 2013 в 16:32 0

Про утиную типизацию сказано в подразделе «Обобщенное программирование», просто не указано, что эту возможность принято называть утиной типизацией.

 **steck** 4 декабря 2012 в 11:37 +1

> Однако автоматический вывод типов довольно сложная вещь, и даже в таком крутом языке как Haskell, он иногда не справляется. (как пример можно привести ограничение мономорфизма)

Ограничение мономорфизма используется для оптимизации, а не по причине причине несправляемости.
Например, представим что у нас есть функция.

```
test xs = let l = length xs in (l, l)
```

То есть функция принимает на вход массив и выдаём массив и выдаёт пару (длина массива, длина массива)

Если есть ограничение мономорфизама, то у длины будет выведен тип
`test :: [a] -> Int`
и всё посчитается, как интуитивно ожидалось.

А теперь представим, что ограничения нет и используется максимально общий тип.
`test :: (Num b, Num c) => [a] -> (b,c)`
то есть функция принимает на вход массив и выдаёт два «числа». Два объекта, у которых есть все свойства чисел.

Пусть у нас есть помимо Int ещё представитель TraceInt, который каждое своё действие (сложение, умножение и тп) пишет в лог.
тогда, можно написать

```
let use1 = test [1] :: (Int, TraceInt)
```

Поскольку никакого преобразования из TraceInt в Int в принципе не существует, то длинну списка придётся вычислять дважды.
Это тоже понятно и логично. Проблемы начинаются когда мы пытаемся использовать тот же тип.

```
let use2 = test [1] :: (Int, Int)
let use3 = test [1] :: (TraceInt, TraceInt)
```

Компилятор не может заранее знать как будет вызываться метод в будущем, поэтому в данных примерах также вычисляет длинну

списка по 2 раза. То есть в выражении
`test xs = let l = length xs in (l, l)`
будет больше вычислений, чем ожидается по здравому смыслу.

Итог: если не использовать ограничения мономорфизма и всегда стараться брать максимально общий тип, то возможны огромные просадки производительности.

atd 4 декабря 2012 в 13:20 0

Свежие версии C# я бы отнёс к полу-явной типизации из-за type-inference (var). А F# по способностям выведения типов уже почти подобрался к хаскеллу.

Throwable 4 декабря 2012 в 15:18 +5

> Тем-не менее не бывает языков со статической и динамической типизаций одновременно. Хотя забегая вперед скажу, что тут я вру — они действительно существуют, но об этом позже.
Groovy имено такой язык. Не просто поддерживает динамику, а именно задумывался как язык одновременно со статической и динамической типизацией.

> Преимущества статической типизации...
Основное и главное (для меня) преимущество статической типизации — это поддержка IDE. Не нужно при вызове API каждый раз бегать в документацию, сверять имена и типы параметров. IDE сама показывает необходимые варианты, исходя из контекста. В итоге работа со сторонними API убыстряется в разы.

Fai 4 декабря 2012 в 15:33 0

> Основное и главное (для меня) преимущество статической типизации — это поддержка IDE.

Спасибо, хороший пункт, добавлю в статью.

kost_bebix 4 декабря 2012 в 15:59 0

Вот, рекомендую:

What To Know Before Debating Type Systems blog.steveklabnik.com/posts/2010-07-17-what-to-know-before-debating-type-systems

amosk 4 декабря 2012 в 16:40 −1

> 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.
> ...
> Примеры:
> Сильная: Java

И что, в джаве так нельзя написать?

```
System.out.println(1 + "a");
```

НЛО прилетело и опубликовало эту надпись здесь

amosk 4 декабря 2012 в 16:58 −1

В статье речь идет про типы вообще.
int — тип
String — тип
1 + «a» — выражение с различными типами

Какая разница какие там свойства у String если данный пример противоречит приведенному в статье свойству сильной типизации — «не позволяет смешивать в выражениях различные типы»

senia 4 декабря 2012 в 17:13 +2

Тут не совсем такая ситуация.
Тут оператор (не метод!) "+".
Он просто определен таким образом, что принимает с одной стороны строку, а с другой — Object. Здесь не выполняется преобразование.

Хотя да, вы можете вспомнить про boxing/unboxing.

amosk 4 декабря 2012 в 17:52 0

Да нет здесь никакой другой ситуации.
Сроки — это часть языка, у них есть тип.
Вы не можете переопределить оператор + для других типов.
Возможность такого сложения для строк предоставляется не библиотекой, а встроенной системой типов джавы во время компиляции, и является ничем иным как неявным преобразованием типов.

senia 4 декабря 2012 в 18:05 +2

> Вы не можете переопределить оператор + для других типов.

Java вообще не дает доступа к операторам. Их нельзя переопределять и добавлять новые.

Любой оператор в Java — это скорее синтаксическая конструкция.
Имело бы смысл говорить о преобразовании типов, если бы был класс мест, в которых это преобразование бы выполнялось. Например если бы в метод, принимающий строку, можно было передать что угодно с неявным преобразованием к строке.

Если хотите, конкатенация в Java — синтаксический сахар над созданием StringBuilder, несколькими вызовами append и затем toString.

И здесь не будет неявного преобразования к строке. Здесь будет вызов append(Object obj) (у вас append(int i)) у StringBuilder.

1 + «a» + someObject — это StringBuilder().append(1).append(«a»).append(someObject).toString().

amosk 4 декабря 2012 в 18:44 0

Да не надо мне рассказывать как это внутри реализуется — я это отлично знаю.
Внутри, в генерируем коде, может быть все что угодно. Например, следуя вашей логике возражений, в джаве в каком то смысле вообще нет никаких типов, классов и методов, а есть интерпретатор и куски бинарных данных которые интерпретируются.

Неявное преобразование типов это вообще-то и есть синтаксический сахар.
То что у джавы оно допускается не везде, не означает что его там нет.

Но на самом деле мое возражение связано с некорректностью утверждения что при сильной типизации недопускаются выражения с разными типами. Оно по сути бессмысленно, что подтвержается моим примером.

senia 4 декабря 2012 в 18:57 +2

Согласен, что в татье дано не идеальное определение сильной типизации.
Не согласен с тем, что в java есть неявное преобразование типов.
Пример с оператором "+" — это не неявное преобразование. Это преобразование не менее явное, чем при вызове append(Object obj).

НЛО прилетело и опубликовало эту надпись здесь

amosk 4 декабря 2012 в 17:41 0

Вы хотите сказать что класс не является типом в том смысле про который идет речь в концепции типизации?

senia 5 декабря 2012 в 11:07 0

Честно говоря до сих пор не понял ваш ответ.
При чем тут System.out.println и java.io.PrintStream.println, если результат выражения {1 + «a»} — String?

Судя по всему @amosk акцентирует внимание именно на операторе "+" и его действии.

Если рассматривать "+" как математическую операцию, то это операция вида (A, A) → A. То есть переводящая пару объектов из **одного** множества в объект из того же множества.
При таком подходе эта операция не определена даже для пары (0, 0.0), так как это объекты разных множеств (разных типов) и требуется неявное преобразование.
То же самое и в случае (_, String) -> String. Первый аргумент, при математическом рассмотрении, неявно преобразуется к String.

Выше я пытался аргументировать к тому, что "+" в Java не соответствует своему математическому тезке, а представляет отдельную строго определенную операцию.

Ваши же аргументы я понять не могу. Мне даже не понятно часть с апеллированием к примитивным типам в Java, так как они в языке, позиционирующемся как чистый ООП, представляют из себя отход от основной идеологии в целях быстродействия.

НЛО прилетело и опубликовало эту надпись здесь

gleb_kudr 4 декабря 2012 в 18:02 +1

Это по сути перегрузка арифметического оператора. К типизации отношение имеет косвенное.

jetman 4 декабря 2012 в 20:39 +1

Жаль, что Go не рассмотрели. В первую очередь интересен тем, что он новичок (по дате создания) в сравнении с популярными языками программирования.

Lendon 4 декабря 2012 в 22:00 0

А, кстати, тот же Go имеет неявную типизацию и при этом в сигнатурах методов указываются типы возвращаемых значений.

BlackRaven86 4 декабря 2012 в 23:27 0

Для C++ есть еще QVariant из Qt, он позволяет некоторые вещи делать как в динамических языках. Особенно это нужно при интеграции C++ кода с кодом на QML.

Fil 8 декабря 2012 в 20:04 +2

Я правильно понял, что если язык имеет динамическую типизацию, то он имеет и неявную?

Fai 8 декабря 2012 в 20:08 +3

Я не знаю языков с динамической явной типизацией (что подтверждает Ваш вывод), но чисто теоретически они могут существовать.

Fil 8 декабря 2012 в 20:26 +1

Гм. Объявляем явно, а потом присваиваем, что хотим :) Хотя, если рассматривать полиморфизм как ограниченную динамическую типизацию, то можно представить и такое. Но это я за уши притягиваю :)

Fai 8 декабря 2012 в 20:32 +1

Ну можно немного подругому сделать, поглядите:

Язык с динамической неявной типизацией:

```
var x = 1;

if (something)
    x = 'Hello!';
else
    x = Complex(1, 3);
```

Язык с динамической явной типзацией

```
var x: Integer = 1;

if (something)
    x: String = 'Hello!';
else
    x: Complex = Complex(1, 3);
```

Таким образом можно перенести в динамическую типизацию некоторые преимущества статической (например `x: String = 3;` компилироваться/интерпретироваться не будет), т.е. программист должен будет явно указывать когда хочет запихнуть в переменную значение другого типа, что можно использовать и для проверки ошибок, и для оптимизации.

А вообще, все это — proof of concept.

Fai 8 декабря 2012 в 20:34 0

И стоит заметить, что несмотря на то, что мы с таким языком получаем некоторые плюсы статической типизации, при этом теряется большая часть (но не все) динамической.

Fil 8 декабря 2012 в 20:46 0

Любопытно. Но, действительно, появляются проблемы. Есть над чем поразмышлять.

alexanderzaytsev 10 января 2013 в 12:33 0

Это ж ~~бубльгум~~Visual Basic 6

grossws 27 августа 2016 в 03:32 0

Кстати такая ситуация может наблюдаться в rust. Правда, там не переменные, а биндинги, поэтому под одно и то же имя может соответствовать разному типу.

Вполне допустимы ситуации типа:

```
let a: &str = "42";

// explicit type in let
let a: i32 = a.parse().unwrap();

// explicit type in parse call
let b = "18".parse::<i32>().unwrap();

// compile-time error, type cannot be inferred
// let c = "18".parse().unwrap();

println!("{}, {}", a + 1, b - 1);
```

playground

alexanderzaytsev 10 января 2013 в 12:44 0

C#. Для того чтобы тип был динамическим его нужно явно объявить таковым.

NeoCode 6 марта 2014 в 23:00 0

Структурную и номинативную типизации забыли обсудить.

art_of_press 17 марта 2015 в 16:21 0

> Примечание: я намерено использовал некаррированную функцию, а также намерено записал частную сигнатуру вместо более общей add :: (Num a) => a -> a -> a*, т.к. хотел показать идею, без объяснения синтаксиса Haskell'a.

Если вы используете кортеж, то сигнатура функции будет:

```
add :: (Num a) => (a, a) -> a
```

barmaley_exe 17 марта 2015 в 19:02 0

У некаррированной add такая и указана (со «специализацией» a = Integer).

misha_shar53 29 сентября 2015 в 06:46 −1

Интересная статья. Впервые столкнулся с систематическим изложением данного вопроса. У меня в голове от всего этого была каша. Смешение понятий в вопросе типизации довольно частое явление в статьях. Хочу добавить несколько своих замечаний. Конечно ассемблер безтиповой язык. На уровне языка нет типов. Есть регистры, память и операции над ними. Как то на них отображаются типы языков высокого уровня. Но это уже проблемы этих языков.
В статье хорошо изложены концепции типизации и хорошо видны противоречия возникающие в типизированных языках. Ясно, что необходима и статическая и динамическая и сильная и слабая типизация. И различные языки по разному находят этот баланс. В основном за счет расширения типов в языке или встроенных библиотеках. При этом совершенно очевидно что базовых типов недостаточно и появляются типы вроде даты и IP адресов. В целом все это мне напоминает ситуацию, когда сначала строят стену, а затем ее героически преодолевают. И очень горды там, как это удачно удалось осуществить. Популярный миф о том что типизация помогает создавать более надежные программы, не более чем миф. Все обстоит с точностью до наоборот. Яркий пример тому бестиповой язык MUMPS. Отсутствие в нем типов решает все проблемы типизации. Они в MUMPS просто отсутствуют. В результате

имеется язык который можно освоить за один вечер. При этом простота языка никак не сказалась на его возможностях. Типизация это зло. Конечно картина с типизацией неоднозначна. Типы тесно связаны со структурой данных. Декларирование данных позволяет так же описать структуру данных, например массив. Встроенные библиотеки добавляют другие структуры стеки, очереди, хеш таблицы. Безтиповой язык должен решать и эту проблему. И MUMPS ее блестяще решил. Любая переменная может являться деревом произвольной структуры, в том числе и простой переменной. А уж из дерева сделать массив, стек или хеш таблицу ничего не стоит. Все типы ненужны. Транслятор сам разбирается с типом переменной. Операции однозначно задают тип операндов и результата. Такие понятия как типы переменных можно выбросить из головы и заняться поставленной задачей. Кстати рассматриваемый пример на MUMPS будет выглядеть так:

```
find(.mass,element) set node=»»
for i=0:1 set node=$O(mass(node)) q:node=»»!(node=element)
q i
```

При этом переменная mass может моделировать и массив и стек и хеш таблицу.

VolCh 30 сентября 2015 в 19:45 +1

> Популярный миф о том что типизация помогает создавать более надежные программы, не более чем миф. Все обстоит с точностью до наоборот. Яркий пример тому бестиповой язык MUMPS.

А где примеры надежности?

Throwable 1 октября 2015 в 10:56 +1

Это такой тонкий толлинг? Открываем Википедию и читаем:

Критики MUMPS прямо называют эту технологию устаревшей[3] и указывают на такие недостатки MUMPS как[3][4]:

- **отсутствие типизации (все данные хранятся как строки);**
- низкий уровень абстракции;
- нечитабельность синтаксиса, особенно при использовании сокращений.

Язык MUMPS критики называют провоцирующим ошибки, поскольку[3][4]:

- отсутствует обязательное объявление (декларирование) переменных;
- не поддерживаются привычные приоритеты арифметических операций (например, выражение 2+3×10 даёт в MUMPS значение 50);
- лишний пробел или разрыв строки может совершенно изменить смысл синтаксической конструкции;
- ключевые слова языка не зарезервированы и могут широко использоваться в качестве идентификаторов.

misha_shar53 1 октября 2015 в 17:49 0

>Это такой тонкий толлинг? Открываем Википедию и читаем:
А что Википедия истина в последней инстанции? А для обвинений надо быть в курсе.

>Критики MUMPS прямо называют эту технологию устаревшей[3] и указывают на такие >недостатки MUMPS как[3][4]:
>отсутствие типизации (все данные хранятся как строки);
Это достоинство а не недостаток. А как что хранится неизвестно и даже безразлично. Программиста это никак не касается. Я думаю, что различные реализации MUMPS данные хранят по разному.
>низкий уровень абстракции;
Совершенно бездоказательное утверждение. На чем основан такой вывод? Абстракция от типов точно есть. Ни разу не встречал анализа уровня абстракции языка MUMPS.
>нечитабельность синтаксиса, особенно при использовании сокращений.
Не сокращай. Но даже с сокращениями вполне читабельный синтаксис. Ничем принципиально не отличающийся от других языков программирования.

>Язык MUMPS критики называют провоцирующим ошибки, поскольку[3][4]:
>отсутствует обязательное объявление (декларирование) переменных;
Ну это никак не провоцирует ошибки. Все перевернуто с ног на голову. Само декларирование порождает массу понятий и правил их использования. И как следствие именно оно и провоцирует ошибки.
>не поддерживаются привычные приоритеты арифметических операций (например, >выражение 2+3×10 даёт в MUMPS значение 50);
Да именно так. Непривычно не значит хуже. Приоритеты операций отсутствуют. Это очень простое правило которым легко пользоваться и трудно допустить ошибку. А вы попробуйте запомнить приоритеты всех операций в Си. Голова идет кругом. Я все равно вынужден либо расставлять скобки, либо каждый раз лезть в справочник. Приоритеты в Си не везде очевидны.

>лишний пробел или разрыв строки может совершенно изменить смысл синтаксической конструкции;
ключевые слова языка не зарезервированы и могут широко использоваться в качестве идентификаторов.
Ну и что? Во многих языках пробел завязан в синтаксисе. И как это влияет на надежность программ? MUMS на порядки надежней любого другого языка. Это простой, логичный, компактный и не противоречивый язык. В нем нет подводных камней. Описание всего языка вместе с примерами занимало 20 страниц книжки половинного формата А4. И этого вполне достаточно. При программировании в основном встречается 2 ошибки деление на ноль и неопределенный индекс. Так как нет декларирования и развитая структура переменных, программы просты, компактны и надежны. Я с 1984года программирую на MUMPS и считаю, что то что написано в Википедии поверхностный взгляд некомпетентного специалиста.

 **misha_shar53** 1 октября 2015 в 18:15 0

>Критики MUMPS прямо называют эту технологию устаревшей[3]
А Си, Паскаль, Си++ не устаревшие? А новые, что принципиально лучше старых? Чем это? Новые языки делаются очень просто. Берем Си и добавляем пробел в конце или еще что нибудь. Принципиально нового, кроме MUMPS ничего нет. Только в MUMPS управление данными полностью взял на себя язык. Унифицировано обращение к данным в памяти и на внешних носителях. В MUMPS работать с встроенной базой данных так же легко как и с памятью. То что я на MUMPS писал месяц, потом на Delphi переносил 2 года при этом пришлось купить и прочитать целый стеллаж книг. Программирование не на MUMPS это убийство времени, ресурсов и сил.

 **grossws** 27 августа 2016 в 02:39 0

> Ruby — Динамическая | Сильная | Неявная

Не соглашусь про сильную (если определять её как отсутствие автоматического приведения вообще). В ruby есть coercion, который позволяет автоматически приводить типы (расширять по необходимости, например, целые в числа с плавающей точкой).

Аналогично в случае java возможно автоматическое приведение типа, если не происходит потери данных (widening primititve conversion `byte -> short -> int -> long -> float -> double`, `char -> int -> long -> float -> double`) или если тип переменной/параметра является суперклассом приводимого объекта. Плюс есть numeric promotions аналогичные механизму coercion в ruby, которые автоматически расширяют типы в арифметических выражениях при необходимости. Более подробно см. 5 главу JLS.

Это даже не вспоминая про boxing/unboxing, которые тоже являются неявной конверсией, которая может приводить к `OutOfMemoryError` или `NullPointerException`.

 **grossws** 27 августа 2016 в 02:47 0

Дополню, что C++ с аналогичной Java ситуацией, касающейся приведения типов, отнесён к слабой. А ведь там даже `void *` неявно не приводится к любому указателю в отличии от C (хотя в g++ это можно разрешить с помощью `-fpermissive`).

 **DmitryKoterov** 19 июля 2019 в 09:48 0

Привет из 2019-го!
Из новостей — появился TypeScript.

 **Maksclub** 15 декабря 2019 в 11:07 +1

PHP тут в конце списков и представлен как самый-самый неявный и самый слабый в плане типизации.

НО! с 2017 года он сильно скаканул и курс на типизацию сильно вырос. Хотя он еще разрешает работать в такой манере — во многих компаниях (во всех, где я работал) курс взят на написание более строго кода.

По поводу неявности типов в недавнем релизе PHP 7.4:

```php
<?php
declare(strict_types=1);

class User
{
     private Id $id;
     private string $name;

     public function rename(string $name): User
     {
         $this->name = $name;

         return clone $this;
     };
}
```

Такой код, очевидно каждому, не назовешь неявно типизированным :)
За строгость тут отвечает declare(strict_types=1), которая не даст приводить скаляры неявным образом там, где указан тип

 **VolCh** 15 декабря 2019 в 17:36 +2

Динамичечкая типизация местами сильная. Причём не строгая, даже со strict_types, поскольку есть неявные преобразования int float как минимум.

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

Не говорите «I feel myself», и другие правила английского языка, которые вгоняют в ступор

+27 12,8k 115 42

Сооснователь Qiwi предоставит Дурову кредит

+18 10,9k 2 23

Инсайды от сотрудника Facebook: как попасть на стажировку, получить оффер и все о работе в компании

+32 9,4k 59 67

Нападение на удаленку

+39 12,5k 37 4

Опять про «MikroTik» или долгожданный SOCKS5

+25 12,7k 52 51

Если нашли опечатку в посте, выделите ее и нажмите Ctrl+Enter, чтобы сообщить автору.