Поддержка типа dynamic в JVM

Алексей Степанов

Научный руководитель: Андрей Бреслав

САНКТ-ПЕТЕРБУРГСКИЙ АКАДЕМИЧЕСКИЙ УНИВЕРСИТЕТ

24 апреля 2017 г.

Введение

Типизация в языках программирования

- Статическая типизация
- Динамическая типизация
- Постепенная типизация

Введение

Постепенная типизация

Постепенная типизация - система типов, в которой часть переменных и выражений может быть типизированна, и их корректность проверяется в момент компиляции, а часть может быть не типизированна, и об ошибках в них мы узнаем во время исполнения.

Преимущества

- eval
- DSL (Gradle)
- DOM

Удобный доступ к полям и методам

```
dynamic x = dom.html.body.tables.main.tr.td;
```

Почему не Object?

```
Scriptobj.SetProperty("Cnt",((int)GetProperty("Cnt"))+1);
scriptobj.Cnt += 1;
```

Варианты постепенной типизации

От динамической типизации	От статической типизации
Python	C#
JavaScript	Scala
Groovy	Kotlin?

Python: Type Hints

```
def greeting(name: str) -> str:
    return 'Hello ' + name

AnyStr = TypeVar('AnyStr', str, bytes)
def concat(x: AnyStr, y: AnyStr) -> AnyStr:
    return x + y
```

Scala

```
class MyRouter extends Dynamic{
    def selectDynamic(name: String): T = {}
    def updateDynamic(name: String)(value: T): Unit = {}
    def applyDynamic(methodName: String)(args: Any*) {
       println(s"You called '$methodName' method with " +
          s"following arguments: ${args mkString ", "}")
     }
     def applyDynamicNamed(name: String)
                          (args: (String, Any)*) {
        println(s"You called '$name' method with " +
            s"following argiuments:
            ${args map (a=>a._1+"="+a._2) mkString ","}")
    }
}
```

Scala

+ Поддержка в ScalaJS
+ Возможность написать свою логику диспетчеризации
– Нет стандартных реализаций
– Своя логика будет скорее всего медленная

C#

```
dynamic obj = new MyObject();
obj.anyMethod(53);
```

Своё разрешение как в Scala

```
public class MyDynamicImpl : DynamicObject{
   public override bool TryGetMember(
        GetMemberBinder binder, out object result){}
   public override bool TrySetMember(
        SetMemberBinder binder, object value){}
}
```

Вопросы постепенной типизации

Возникающие вопросы

- Где хранить информацию о типе?
- Как хранить динамически типизированный объект?
- Как выбирать перегрузки при вызове методов?
- Как сделать всё оптимально и быстро?

- В динамических языках возможна перегрузка только по числу параметров
- В статически типизированных языках у нас доступна перегрузка по типам аргументам

Возникающий вопрос

- Можем ли мы при постепенной типизации делать во время исполнения всё то же что мы делали во время компиляции?

- В динамических языках возможна перегрузка только по числу параметров
- В статически типизированных языках у нас доступна перегрузка по типам аргументам

Возникающий вопрос

- Можем ли мы при постепенной типизации делать во время исполнения всё то же что мы делали во время компиляции?
- Ответ: Нет.
  - Это дорого
  - Это невозможно

Groovy

1. Получить список всех методов с подходящим именем
2. Удалить все методы которые не подходят к данному вызову
3. Если осталось больше одного метода, то вычислить метрику на на методах
4. Метод с наименьшим значением метрики выигрывает
5. Если имеется несколько методов с наименьшей дистанцией, то выдать исключение

Groovy

```
static foo(Object o, Integer i) { "Integer won" }
static foo(String s, Object o) { "String won" }

assert foo("potato", new Integer(6)) =~ "Integer"
```

Полученная дистанция

- $\rho(\text{String}, \text{Object}) + \rho(\text{Integer}, \text{Integer}) == 1 + 0 == 1$
- $\rho(\text{String}, \text{String}) + \rho(\text{Integer}, \text{Object}) == 0 + 2 == 2$
  $(\text{Integer} \to \text{Number} \to \text{Object})$

C#

- Во время компиляции мы проверяем что во время исполнения может подойти хотя бы один метод
- По алгоритму похожему на используемый во время компиляции определяем лучший метод
- Разницу между Object и dynamic не разрешаем

Критерии определения правил выбора перегрузок

- Предсказуемость.
- Производительность.
- Обратная совместимость при стирании типа в статически типизированном коде.
- Схожесть работы нетипизированного кода с типизированным.

Предварительная обстановка

Kotlin

- Поддержка dynamic в Kotlin для JavaScript.
- В JVM dynamic не поддерживается.

Цель и задачи

Цель

Для обеспечения поддержки постепенной типизации в языке Kotlin, возникает необходимость реализовать поддержку типа dynamic при компиляции на Java платформе.

Задачи

- Определить поведение динамических операций.
- Выработать правила разрешения перегрузок.
- Реализовать поддержку динамических вызовов в компиляторе языка Kotlin под JVM.
- Оценить производительность.

Kotlin

- Присваивание в динамическую переменную.
- Присваивание динамической переменной в типизированную.
- Вызов метода на динамической переменной.
- Вызов метода на типизированной переменной.
- Запрос поля у динамической переменной.
- Запрос динамического поля у не динамической переменной.

Алгоритм определения перегрузок методов

- Его имя совпадает с динамически вызванным методом.
- К его аргументам подходят аргументы времени выполнения у динамического метода.
- Он является более специфичный, чем все другие методы, которые удовлетворяют пунктам 1-2.

Реализация

Поддержка в компиляторе

- invokedynamic.
- MethodHandles.
- Выполнение составного присваивания.

  a += b
- Вызываемые объекты

  obj.foo(args)

Сравнение производительности

- Были написаны тесты с использованием JMH.
- Быстрее чем Groovy (обычный) до 14 раз.
- Быстрее чем Groovy (invokedynamic) до 6 раз.
- В некоторых тестах наблюдается ухудшение производительности обладающее хуже чем линейной зависимостью.

Вопросы?

Реализация

Про два типа ϕ и ψ, будем говорить:

- Что ϕ и ψ эквивалентные типы, если они в точности совпадают.
- Что ϕ и ψ похожие типы, если один из них, является упакованной версией другого.
- Что ϕ лучший тип чем ψ, если ϕ реализует интерфейс ψ, или является его потомком.
- Во всех других случаях, будем говорить, что ϕ худший тип, чем ψ.

Реализация

Определение перегрузок

1. Если f совпадает с g, то он более специфичный чем g.
2. Если f является методом-помощником, то он более специфичный чем g.
3. Если g является методом-помощником, то он менее специфичный чем f.
4. Если возвращаемый тип f лучше, чем тип g, то f более специфичный.
5. Если возвращаемый тип f хуже, чем тип g, то f менее специфичный.
6. Если у f существует такой индекс i, что i-ый параметр f хуже чем i-ый параметр g, то f менее специфичный чем g.
7. Если у f существует такой индекс i, что i-ый параметр f лучше чем i-ый параметр g, то f более специфичный чем g.
8. Если g является методом с переменным числом аргументов, а f — нет, то f более специфичный.
9. Во всех остальных случаях, f менее специфичный.