Парадигмы программирования

Чепелин Вячеслав

Содержание

1 Java

1.1 Программирование по контракту

Вот у нас есть функция:

```java
public class Contracts{
    int magic(int a, int n){
        int r = 1;
        while(n != 0){
            if(n % 2 == 1){
                r *= a;
            }
            n /= 2;
            a *= a;
        }
        return r;
    }
}
```

У нас есть предположение, что эта функция делает бинарное возведение в степень. Но мы в этом не уверены. Мы внимательно посмотрели на код и поняли, что вот это и правда возведение в степень. Но в реальности такое понимать довольно сложно. Сегодня мы поднимемся на уровень выше. Мы научимся доказывать, что наша программа делает именно то, что она должна на понятном математическом языке. Мы будем смотреть на программу, как на математический объект. Мы воспользуемся **тройкой Хоара**.

В терминологии Хоара у нас есть:

- C - блок кода со входом и выходом.
- $\{P\}$ - пред условие - то, что должно выполняться, чтобы исполнить.
- $\{Q\}$ - пост условие - то, что гарантирует блок C в результате того, что он был исполнен.

Возникает вопрос. Какую структуру может иметь промежуточный код?

Теорема Бёма - Якопини

Если есть какой-то блок кода, то его можно переформулировать в следующую конструкции:

- последовательное исполнение.
- присваивание.
- ветвление.
- цикл while.

Тяжелую концептуальную вещь мы можем преобразовать в такую штучку. А теперь применим к этому тройку Хоара:

Последовательное исполнение: У меня есть 2 подряд идущих блока кода: $\{P_1\}C_1\{Q_1\}$, $\{P_2\}C_2\{Q_2\}$. Какое у них будет пред и пост условие? Кто-то скажет, что Pred: P_1 Post: Q_2

Достаточно ли этого? Нет. Мы не можем гарантировать P_2 в таком случае. Чтобы гарантировать это, мы также хотим, чтобы из Q_1 следовало P_2.

Присваивание: $x = expr$ - обычное присваивание. Есть пред. условие P, что мы можем сказать, про пост условие? Тут не все так тривиально. Если мы хотим пост условие Q, то я должен потребовать в начале $Q[x \to expr]$. Примеры:

```
{ a + 1 < b }
x = a + 1
{ x < b }
```

```
{ a + 1 < b }
x = a + 1
{ x < b }
```

То есть такой подход с подстановкой делает то, что от нас требуется.

Ветвление:

```
if(cond) {
 { P1 } C1 { Q1 }
}else{
 { P2 } C2 { Q2 }
}
```

Давайте подумаем, какое у нас пост условие:
Post: Q1 || Q2 - такое решение очень странное так как мы вообще не шарим, что там будет. Как код писать то следующий? Следуюший код будет делать еще ветвление, а зачем нам его еще раз делать?
Так что введем Q такую, что $Q1 \to Q$, $Q_2 \to Q$.

Теперь поговорим про то, что будет в пред условии:
Нам нужно, чтобы из P && cond $\to P1$, а так же P && !cond $\to P2$.

Такой подход делает то, что от нас требуется:

While:

```
while(cond){
 { Pi } Ci { Qi }
}
```

Посмотрим на самый первый заход в в цикл.
Заметим, что из P && cond$\to Pi$. Пусть $Qi \to P$ - можем исполнять сколько угодно раз. Теперь посмотрим на пост условие - P && !$cond$.

Давайте докажем работу функции из начала:

```
// Pred: n >= 0
// Post: R = a ^ n
int power(int a, int n) {
    int r = 1;
    // r' * a' ^ n' = a ^ n
    while (n != 0) {
        // I && n' != 0
        if (n % 2 == 1) {
            // I && n' % 2 = 1
            r *= a; n--;
            // I && n' % 2 = 0
        } else {
            // I && n' % 2 = 0
        }
        // I && n' % 2 = 0
        n /= 2; a *= a;
        // I
```

```
    }
    // r' * a' ^ n' = a ^ n && n' = 0
    // => r' = a ^ n
    // => R = r'
    return r;
}
```

Функции

Чистые функции

- Результат зависит только от аргументов

- Не имеет побочных эффектов (не меняют ничего снаружи самой себя)

Предусловие — условие, которое должно быть верно на момент вызова. Результат вызова с неверным предусловием не определен.

Постусловие — условие, которое верно на момент возврата. Если постусловие не выполнено, то в программе есть ошибка.

Например:

```
// Pred: x > 0
// Post: R * R = x ^ R >= 0
double sqrt(double x) {
    ...
}
```

Функции с состоянием

```
// Состояние
int value = 0;

// Pred: v >= 0
// Post: R = value + v && value' = value + v
int add(int v) {
    return value += v;
}

```

Добавим пред и постусловия на неотрицательность ‘value‘:

```
// Состояние
int value = 0;

// Pred: v >= 0 && value >= 0
// Post: R = value + v && value' = value + v && value >= 0
int add(int v) {
    return value += v;
}
```

Инвариант — общая часть пред и постусловия. Выполняется всегда. Обозначается как ‘Inv‘.

```
// Inv: value >= 0
int value = 0;

// Pred: v >= 0
// Post: R = value + v && value' = value + v
int add(int v) {
    return value += v;
}
```

Инвариант + предусловие + постусловие функции с состоянием называется *контрактом*.

2 JavaScript

2.1 Введение в JavaScript

> Если вы на практике хотите сравнивать NaN и undefined, то очень жаль, что вам до этого кто-то голову не открутил.
>
> Георгий Корнеев

2.1.1 О языке:

- Изначально создан для добавления интерактивности веб-страницам. Сейчас используется гораздо шире: для разработки веб-приложений (frontend и backend), мобильных приложений, десктопных приложений, игр и многого другого.
- Официально язык стандартизирован как ECMAScript (ECMA-262). JavaScript - это наиболее распространённая реализация стандарта ECMAScript.
- Таким образом, строго говоря, "ECMAScript это спецификация языка, а "JavaScript конкретная реализация этой спецификации. Но на практике эти термины часто используются как взаимозаменяемые.

2.1.2 Как пишется код? Основные вещи.

Типы данных и объявление переменных:

Переменные объявляются с помощью ключевого слова let:

```
let v = 10;
```

С помощью typeof мы можем более точно понять, что за тип:

```
>> a = 3
>> typeof(a)
"number"
>> a = hello
>> typeof(a)
"string"
>> a = [1, 2, 3]
>> typeof(a)
"object"
```

На следующей лекции мы узнаем, что такое object, пока что забьем.

Так же в Javascript есть не только null, но и undefined. Смысл у этого такой, что:

```
let a = null; у переменной есть значение и это ничего  === null
let b; у переменной нет значения === undefined
```

Веселые места в javascript (оператор равенства):

> Не надо интерпретировать массивы как числа и у вас все будет хорошо!
>
> Георгий Корнеев

Давайте рассмотрим некоторые **приколы**:

```
>> "1" == "1"
true
```

Ну это звучит максимально логично, пока мы не узнаем, что:

```
>> "1" == 1
true
>> "1" == 1.0
true
>> "1.0" == 1
true
```

Бывают более интересные вещи:

```
>> 0 == []
true
>> undefined  == null
true
>> 10 == [10]
true
```

Для этого в JavaScript есть 2 типа сравнения: приводящее (‘==‘) и неприводящее (‘===‘). Приводящее приводит данные к одному типу, а затем сравнивает, а неприводящее выдаёт ‘false‘, если типы не равны.

Типизация.

Во-первых стоит разделять 2 вида типизации:

1. **статическую** - во время компиляции.
2. **динамическую** - во время исполнения.

Статической типизации в JavaScript **нет**. В Javascript слабая динамическая типизация.

Массивы.

Посмотрим на примерах, они крайне подробно описывают происходящее:

```
>> a = [10, 20, 30];
>> a.push(40,50);
>> dump(a);
length: 5, elements: [10, 20, 30 , 40, 50]
>> a.pop();
50
>> a.pop();
40
>> a.unshift(100); - вставить в начало, никто не знает за сколько это работает.
 ↪  Приличные люди не задают такие вопросы, особенно JavaScript-у
```

```
>> a.shift();
100
>> a[10] = 10;
>> dump(a);
length: 11, elements: [10, 20, 30, , , , , , , , 10]
>> a[5]
undefined
>> a["hello"] = "world"
>> a[-1] = "???"
>> dump(a)
length: 11, elements: [10, 20, 30, , , , , , , , 10], other{-1 = ???, hello = world}
>> as[as]  = as
length: 11, elements: [10, 20, 30, , , , , , , , 10], other{-1 = ???, hello = world, 10,
↪  20, 30, , , , , , , , 10 = 10, 20, 30, , , , , , , , 10 }
>> a.pop();
10
>> a.dump();
length: 11, elements: [10, 20, 30, , , , , , , , 10], other{-1 = ???, hello = world, 10,
↪  20, 30, , , , , , , , 10 = 10, 20, 30, , , , , , ,}
```

Как видно массивы в JavaScript работают превосходно!

Константы

```
>> const c = 10
>> c = 20
Uncaught TypeError: Assignment to constant variable.
```

Константы нельзя обновлять. Но например, если это будет объект, то внутри него можно будет менять значения.

2.1.3 Функции.

Функции - это переменные:

```
let dumpArgs = function(){
    println(typeof(arguments));
    println(arguments.constructor.name);
}
>> dumpArgs(10, "hello", null);
object
Object  --- иногда Arguments
```

Вот так создаются функции. Arguments ведет себя как массив:

```
dumpArgs = function() {
    for (const v of arguments) {
        println(v);
    }
}
>> dumpArgs(1, 2, "hello", null)
1
2
```

```
hello
null
```

Вообще у нас const v - неизменняя переменная. Слово переменная говорит о том, что она меняется. А модифактор const говорит, что он не может. Задумайтесь :)

```
sum = function() {
    let s = 0;
    for (const v of arguments) {
        s += v;
    }
    return s;
}
>> sum(1, 2, 3)
6
```

У javascript есть:

```
use strict
```

Он запрещает производить некоторые совсем извращенские штуки.

Ещё функции можно объявлять так:

```
function min(a, b) {
    return a < b ? a : b;
}
>> min(3, 4)
3
```

Но это не означает, что мы не можем вызывать функцию с другим количеством аргументов:

```
>> min(4)
undefined
>> min(1, 2, -10)
1
```

Формально он просто именует первые n аргументов (в данном примере 2).

Можно положить все остальные аргументы в массив:

```
 function min(first, ...rest) {
    for (const v of rest) {
        if (v < first) {
            first = v;
        }
    }
    return first;
}
>> min(1, 2, 3, -10, 5)
-10
```

После rest параметра ничего быть не может.

Второй вид функций —стрелочные функции.

```
let min = (first, ...rest) =>{
    for (const v of rest) {
        if (v < first) {
            first = v;
        }
    }
    return first;
}
```

Существенное отличие — все аргументы именованные.

Можно писать еще:

```
const add = (a, b) => a + b.
```

Функции в аргументы к функциям. Легко!

Функции можно передавать в аргументы другим функциям и возвращать как результат! Такие функции называются **функциями высшего порядка**:

```
minByComparator = (compare, init = Infinity) => {
    return (...args) => {
        let result = init;
        for (const v of args) {
            if (compare(result, v) > 0) {
                result = v;
            }
        }
        return result;
    };
}
>> minBy((a, b) => a - b)(10, 20, -30, 40)
-30
>> comparing = (f) => ((a, b) => f(a) - f(b))
>> minByAbs = minBy(comparing(Math.abs))
>> minByAbs(10, 20, -30, 40)
40
>> regularMin = minBy(compare(identity))
>> regularMin(10, 20, -30, 40)
-30
```

Ещё пример — **левая свёртка**. Может создать функцию для суммы, максимума, минимума и любой ассоциативной операции, применяемой на множество аргументов:

```
foldLeft = (f, zero) => {
    return (...args) => {
        let result = zero;
        for (const v of args) {
            result = f(result, v);
        }
        return result;
    };
}
```

```
>> sum = foldLeft((a, b) => a + b, 0)
>> sum(1, 2, 3)
6
```

При помощи функций высшего порядка можно сделать универсальную функцию для нахождения производной:

```
diff = dx => f => x =>(f(x + dx) - f(x - dx)) / (dx * 2)
>> dSin = diff(1e-7)(Math.sin)
>> dsin(Math.Pi)
-1
```

Ещё пример — **map**. Позволяет применять функцию на все элементы массива:

```
map = (f) => {
    return (...args) => {
        const result = [];
        for (const arg of args) {
            result.push(f(arg));
        }
        return result;
    };
}
>> add10 = map((a) => a + 10)
>> add10(10, 20, 30)
[20, 30, 40]
```

Ещё полезная функция — curry. Позволяет передавать аргументы друг за другом, вместо всех сразу:

```
>> curry = f => a => b => f(a, b)
>> add = curry((a, b) => a + b)
>> add10 = add(10)
>> add10(20)
30
```

2.2 Объекты

```
>> let p = Object.create({})
>> p[x] = 10
>> p[y] = 20
>> dumpObject(p)
Object
x = 10
y = 20
```

Можно обращаться p.x и так далее, есили это корректный Java идентификатор. У объекта могут не определены

```
>> println(p[z])
undefined
```

```
for (let prop in p){

}
```

Перебирает свойства и печататет их значения

3 Информация о курсе

Поток — y2024.
Группы M3138-M3139.
Преподаватель — Корнеев Георгий Александрович.