Парадигмы программирования

Альжанов Леонид

27.02.2025

Содержание

Программирование по контракту

Рассмотрим данную функцию:

```
int power(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. Давайте 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 **тройками Хоара**.

В терминологии Хоара у нас есть:

- Блок кода C.
- Предусловие P — то, что должно выполняться, чтобы исполнить.
- Постусловие Q — то, что гарантирует блок C в результате того, что он был исполнен.

Обозначается в коде вот так:

```
// Pre: P
C
// Post: Q
```

Теорема Бёма - Якопини

Она гласит, что любой управляющий граф (любой код программы) можно составить из:

- Присвоения.
- Последовательного исполнения.
- Ветвления.
- Циклов while.

Присвоение

Пусть есть какое-то выражение expr и предусловие $P[x \to \text{expr}]$ (условие от x, где оно заменено на x). Тогда постусловием к $x := \text{expr}$ будет P.

```
// Pre: P[x -> expr]
x = expr;
// Post: P
```

Примеры:

```
// Pre: b = 0
a = b;
// Post: a = 0
```

```
// Pre: a + a = b
a += a;
// Post: a = b
```

Последовательное исполнение

Из:

```
// Pre: P1
S1
// Post: Q1
```

```
// Pre: P2
S2
// Post: Q2
```

$Q1 \Rightarrow P2$

Следует:

```
// Pre: P1
S1
S2
// Post: Q2
```

Ветвление

Из:

```
// Pre: P && cond
S1
// Post: Q
```

```
// Pre: P && !cond
S2
// Post: Q
```

Следует:

```
// Pre: P
if (cond) {
    S1
} else {
    S2
}
// Post: Q
```

Цикл

Из:

```
// Pre: P && cond
S
// Post: P
```

Следует:

```
// Pre: P
while (cond) {
    S
}
// Post: P && !cond
```

В данном случае условие P называется инвариантом цикла. Но нужно доказать, что цикл завершится.

Давайте докажем работу функции из начала.

```
// Pre: 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:

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

Функции с состоянием

```
// Состояние
int value = 0;

// Pre: 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;

// Pre: 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;

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

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

Реализация ООП

Давайте напишем структуру данных стек.

Описание

- Переменные
  - `size` — число элементов
  - `elements` — массив элементов
- Методы:
  - `push(element)` — добавить элемент
  - `pop()` — удалить элемент
  - `peek()` — получить элемент на вершине
  - `size()` — число элементов
  - `isEmpty()` — проверка на пустоту

Модель

- Последовательность чисел $a_1, a_2, ..., a_n$. Операции выше проводятся с последним элементом.
- Инвариант:
  - $n \geq 0$
  - $\forall i = 1, ..., n : a_i \neq \text{null}$
- Вспомогательные определения:
  - $\text{immutable}(k) = \forall i = 1, ..., k : a_i{}' = a_i$

Контракт

- `push(element)`:

```
// Pred: element != null
// Post: n' = n + 1 && immutable(n) && a'[n'] = element
void push(Object element)
```

- `pop()`:

```
// Pred: n > 0
// Post: R = a[n] && n = n' - 1 && immutable(n')
Object pop()
```

- `peek()`:

```
// Pred: n > 0
// Post: R = a[n] && n = n' && immutable(n)
Object pop()
```

- `size()`:

```
// Pred: true
// Post: R = n && n = n' && immutable(n)
Object pop()
```

- `isEmpty()`:

```
// Pred: true
// Post: R = n > 0 && n = n'&& immutable(n)
Object pop()
```

Процедурная реализация

Мы показывем, как работает структура стека, но только на одном экземпляре. Процедурная реализация создана для простых задач, но ее трудно поддерживать и расширять для сложных проектов.

```
public class ArrayStackModule {
    private static int size;
    private static Object[] elements = new Object[1];

    private static void ensureCapacity(int capacity) {
        if (capacity > elements.length) {
            elements = Arrays.copyOf(elements, 2 * capacity);
        }
    }

    public static void push(Object element) {
        Objects.requireNonNull(element);
        ensureCapacity(size + 1);
        elements[size++] = element;
    }

    public static Object pop() {
        assert size > 0;
        size--;
        Object result = elements[size];
        elements[size] = null;
        return result;
        // Если мы собрались управлять памятью руками,
        // то освобождать её мы тоже должны руками
    }

    public static Object peek() {
        assert size > 0;
        return elements[size - 1];
```

```
    }

    public static int size() {
        return size;
    }

    public static boolean isEmpty() {
        return size == 0;
    }
}
```

Реализация на структурах

Будем передавать в каждую процедуру экземпляр, с которым происходит действие. Это позволит создать несколько отдельно существующих экземпляров. Это абстрактный класс с явной передачей ссылки на результат. Улучшает связь данных, но все еще не подходит, так как никто не мешает нам производить действия в данной структуре в другом файле(например).

```
public class ArrayStackADT {
    private static int size;
    private static Object[] elements = new Object[1];

    public static ArrayStackADT create() {
        ArrayStackADT stack = new ArrayStackADT();
        stack.elements = new Object[1];
        return stack;
    }

    private static void ensureCapacity(ArrayStackADT stack, int capacity) {
        if (stack.elements.length < capacity) {
            stack.elements = Arrays.copyOf(stack.elements, capacity * 2);
        }
    }

    public static void push(ArrayStackADT stack, Object element) {
        Objects.requireNonNull(element);
        ensureCapacity(stack, stack.size + 1);
        stack.elements[stack.size++] = element;
    }

    public static Object pop(ArrayStackADT stack) {
        assert stack.size > 0;
        stack.size--;
        Object result = stack.elements[stack.size];
        stack.elements[stack.size] = null;
        return result;
    }

    public static Object peek(ArrayStackADT stack) {
        assert stack.size > 0;
        return stack.elements[stack.size - 1];
    }

    public static int size(ArrayStackADT stack) {
```

```
        return stack.size;
    }

    public static boolean isEmpty(ArrayStackADT stack) {
        return stack.size == 0;
    }
}
```

Преобразование в класс (ООП)

По сути классы — синтаксический сахар поверх реализации выше. У каждого метода есть невидимый аргумент `this`, к которому обращаются при обращении к любому полю класса.

Наиболее мощный и гибкий подход для больших и сложных проектов, обеспечивающий инкапсуляцию, наследование и полиморфизм, но требует больше планирования и проектирования.

```
public class ArrayStack {
    private int size;
    private Object[] elements = new Object[1];

    private void ensureCapacity(int capacity) {
        if (elements.length < capacity) {
            elements =
                Arrays.copyOf(elements, capacity * 2);
        }
    }

    public void push(Object element) {
        Objects.requireNonNull(element);
        ensureCapacity(size + 1);
        elements[size++] = element;
    }

    public Object pop() {
        assert size > 0;
        size--;
        Object result = elements[size];
        elements[this] = null;
        return result;
    }

    public Object peek() {
        assert size > 0;
        return elements[size - 1];
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }
}
```

Информация о курсе

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