Функціональний дизайн: принципи, патерни і практики

Роберт С. Мартін

ОЗНАЙОМЧИЙ ФРАГМЕНТ

Переклад — Кирило Горбушко

Що таке функціональне програмування?

Якщо ви запитаєте середньостатистичного програміста, що таке функціональне програмування, ви можете отримати будь-яку з таких відповідей:

Хоча ці твердження можуть виявитися правдивими, вони не є особливо корисними. Я думаю, що краща відповідь буде такою:

Це програмування без операторів присвоювання.

Можливо, це визначення не здається вам набагато кращим. Можливо, воно вас навіть лякає. Зрештою, яке відношення оператори присвоювання мають до функцій, і як узагалі можна програмувати без них?

Хороші запитання. Саме на них я маю намір відповісти в цьому розділі.

Розглянемо наступну просту програму на C:

```
int main(int ac, char** av) {
while(!done())
doSomething();
}
```

Ця програма є основним циклом практично кожної будь-коли написаної програми. Вона буквально говорить: «Роби щось, поки не закінчиш». Навіть більше, у цій програмі немає видимих операторів присвоювання. Чи є вона функціональною? І якщо так, то чи означає це, що кожна будь-коли написана програма є функціональною?

Змусимо цю функцію щось робити. Нехай вона обчислює суму квадратів перших десяти цілих чисел [1..10]:

```
int n=1;
int sum=0;
int done() {
return n›10;
```

```
}
void doSomething() {
sum+=n*n;
++n;
}
void sumFirstTenSquares() {
while(!done())
doSomething();
}
```

Ця програма не є функціональною, тому що вона використовує два оператори присвоювання у функції doSomething. Крім того, ці дві глобальні змінні виглядають просто потворно. Тож покращимо її:

```
int sumFirstTenSquares() {
int sum=0;
int i=1;
loop:
if (i>10)
return sum;
sum+=i*i;
i++;
goto loop;
}
```

Це вже краще; дві глобальні змінні стали локальними. Але це все ще не функціонально. Можливо, вас турбує цей goto. На те є вагома причина. Потерпіть, поки ми розглядаємо цю невелику модифікацію, що використовує робочу функцію для перетворення локальних змінних на аргументи функції:

```
int sumFirstTenSquaresHelper(int sum, int i) {
loop:
if (i>10)
```

```
return sum;
sum+=i*i;
i++;
goto loop;
}
int sumFirstTenSquares() {
return sumFirstTenSquaresHelper(0, 1);
}
```

Ця програма ще не функціонує, але це важлива віха, до якої ми ще повернемося. Але зараз, з останньою зміною, відбувається щось магічне:

```
int sumFirstTenSquaresHelper(int sum, int i) {
if (i›10)
return sum;
return sumFirstTenSquaresHelper(sum+i*i, i+1);
}
int sumFirstTenSquares() {
return sumFirstTenSquaresHelper(0, 1);
}
```

Усі оператори присвоювання зникли, і програма працює. Вона також рекурсивна. І це не випадково. Якщо ви хочете позбутися операторів присвоювання, ви повинні використовувати рекурсію. Рекурсія дозволяє замінити присвоювання локальних змінних на ініціалізацію аргументів функції.

Це дійсно спалює багато місця на стеку. Однак є невеликий трюк, що допоможе розв’язати цю проблему.

Зверніть увагу, що останній виклик sumFirstTenSquaresHelper є також останнім використанням sum та i у цій функції. Тримати ці дві змінні у стеку після ініціалізації двох аргументів рекурсивного виклику безглуздо; вони ніколи не будуть використані. А що як замість того, щоби створювати новий кадр стеку (stack frame) для

рекурсивного виклику, ми просто повторно використаємо поточний кадр стеку, повернувшись до початку функції за допомогою goto, як ми це зробили у віховій програмі?

Цей симпатичний маленький трюк називається оптимізацією хвостових викликів (tail call optimization, TCO), і всі функціональні мови використовують його{8}.

Зверніть увагу, що TCO фактично перетворює цю останню програму на програму етапу. Останні три рядки sumFirstTenSquaresHelper у проміжній програмі є, по суті, рекурсивним викликом функції. Чи означає це, що проміжна програма також є функціональною? Ні, вона просто поводиться ідентично. На рівні вихідного коду ця програма не є функціональною, тому що в ній є оператори присвоювання. Але якщо ми зробимо крок назад і проігноруємо той факт, що локальні змінні змінилися, а не були відновлені в новому кадрі стеку, то програма поводиться як функціональна.

Як ми дізнаємося з наступного розділу, це не є відмінністю без відмінності. А поки що просто пам’ятайте, що коли ви використовуєте рекурсію для усунення операторів присвоювання, ви не обов’язково витрачаєте багато місця у стеку. Мова, яку ви використовуєте, майже напевно використовує TCO.

Проблема із присвоюванням

Спочатку визначимо, що ми маємо на увазі під присвоюванням. Присвоювання значення змінній змінює початкове значення змінної на щойно присвоєне. Саме ця зміна і є присвоюванням.

У C ми ініціалізуємо змінну у такий спосіб:

```
int x=0;
```

Але присвоюємо змінну так:

```
x=1;
```

У першому випадку змінна x починає існувати зі значенням 0; до ініціалізації змінної x не існувало. У другому випадку значення x змінюється на 1. Це може здатися несуттєвим, але наслідки є глибокими.

У першому випадку ми не знаємо, чи є x насправді змінною. Це може бути константа. У другому випадку сумнівів немає. Ми змінюємо x, присвоюючи їй нове значення. Ось чому можна казати, що функціональне програмування — це програмування без змінних. Значення у функціональних програмах не змінюються.

Чому це бажано? Подумайте ось про що:

```
.
//Block A
.
x=1;
.
//Block B
.
```

Стан системи під час виконання Block A відрізняється від стану системи у Block B. Це означає, що Block A має виконуватися перед Block B. Якщо поміняти місцями ці два блоки, то система, найімовірніше, не буде виконуватися коректно.

Це називається послідовне або тимчасове з’єднання — з’єднання у часі; і це те, із чим ви, мабуть, добре знайомі. Open потрібно викликати перед close. New має бути викликаний перед delete. Malloc має бути викликаний перед free. Список таких пар є нескінченним{9}. І багато в чому вони є нашим прокляттям.

Скільки разів ви забували закрити файл або звільнити блок пам’яті, або закрити графічний контекст, або звільнити семафор?

Скільки разів вам доводилося розв’язувати головоломну проблему тільки для того, щоби виявити, що її можна виправити, помінявши місцями два виклики функцій?

А ще є прибирання сміття (garbage collection).

Видалення сміття — це жахливий процес{10}, який ми прийняли в наших мовах, тому що погано керуємо часовими зв’язками. Якби ми вміли стежити за виділеною пам’яттю, ми би не залежали від якогось неприємного фонового процесу, який би прибирав за нами сміття. Але сумний факт полягає в тому, що ми настільки жахливо справляємося з тимчасовими зв’язками, що радіємо милицям, які використовуємо, щоби захистити себе від них.

І це не враховуючи багатопотоковості. Коли два або більше потоків конкурують за процесор, утримання тимчасових зв’язків у правильному порядку стає набагато складнішим завданням. Ці потоки можуть підтримувати правильний порядок 99,99 відсотків часу; але час від часу вони можуть виконуватися і в неправильному порядку і спричиняти пекельний хаос. Ми називаємо такі ситуації умовами перегонів (race conditions).

Тимчасові зв’язки та умови перегонів є природним наслідком програмування зі змінними з використанням присвоювання. Без присвоювання не існує часових зв’язків та умов перегонів{11}. У вас не може бути проблем з одночасним оновленням, якщо ви ніколи нічого не оновлюєте. У вас не може бути проблем упорядкування у функції, якщо стан системи у цій функції ніколи не змінюється.

Мабуть, настав час навести простий приклад. Ось знову наш не функціональний алгоритм; цього разу без goto:

```
1: int sumFirstTenSquaresHelper(int sum, int i) {
2: while (i‹=10) {
3: sum+=i*i;
4: i++;
5: }
6: return sum;
7: }
```

Тепер припустимо, що ви хочете записати хід виконання алгоритму за допомогою такого оператора:

```
log("i=%d, sum=%d", i, sum);
```

Де би ви поставили цю лінію? Є три варіанти. Якщо ви додасте оператор ведення журналу (log) після рядка 2 або 4, то записані дані будуть правильними, і різниця полягатиме лише в тому, чи ви записуєте дані до чи після обчислень. Якщо ви вставите оператор log після рядка 3, то записані дані будуть некоректними. Це і є проблема часового зв’язку — проблема впорядкування.

Тепер розглянемо наше функціональне рішення з однією цікавою косметичною зміною:

```
int sumFirstTenSquaresHelper(int sum, int i) {
return (i>10) ? sum : sumFirstTenSquaresHelper(sum+i*i, i+1);
}
```

Існує лише одне місце, куди ми можемо помістити наш оператор, і він буде реєструвати правильні дані.

То чому ж воно називається функціональним?

Функція — це математичний об’єкт, що перетворює вхідні дані у вихідні. Якщо y = f(x), то для кожного значення x існує значення y. Ніщо інше не має значення для f. Якщо ви передасте x функції f, ви отримаєте y кожного разу. Стан системи, у якій виконується функція, не має значення для f.

Або, інакше кажучи, немає ніяких часових зв’язків з f. Немає ніякого особливого порядку, у якому потрібно викликати f. Якщо ви викликаєте f із x, то отримаєте y незалежно від того, що ще могло змінитися.

Функціональні програми є істинними функціями у цьому математичному сенсі. Якщо ви розкладете функціональну програму на багато менших функцій, кожна з них також буде істинною функцією в тому ж математичному сенсі. Це називається

референтною прозорістю (referential transparency).

Функція є референтно прозорою, якщо ви завжди можете замінити виклик функції її значенням. Спробуймо це зробити з нашим функціональним алгоритмом для обчислення суми квадратів перших десяти цілих чисел:

```
int sumFirstTenSquaresHelper(int sum, int i) {
return (i>10) ? sum : sumFirstTenSquaresHelper(sum+i*i, i+1);
}
int sumFirstTenSquares() {
return sumFirstTenSquaresHelper(0, 1);
}
```

Коли ми замінимо перший виклик sumFirstTenSquaresHelper його реалізацією, він стане:

```
int sumFirstTenSquares() {
return (1›10) ? 0 : sumFirstTenSquaresHelper(0+1*1, 1+1);
}
```

Коли ми замінюємо наступний виклик функції, він стає:

```
int sumFirstTenSquares() {
return
1›10) ? 0 :
(2›10) ? 0+1*1
: sumFirstTenSquaresHelper((0+1*1)+2*2, (1+1)+1);
}
```

Думаю, ви розумієте, до чого це призводить. Кожен виклик помічника sumFirstTenSquares просто замінюється його реалізацією з належно заміненими аргументами.

Зауважте, що ви не можете зробити цю просту заміну нефункціональною версією програми. Так, ви можете розкрутити цикл, якщо хочете; але це не те саме, що просто замінити кожен виклик функції її реалізацією.

Отже, функціональні програми складаються зі справжніх математичних, референтно прозорих функцій. І саме тому це називається функціональним програмуванням.

Без зміни стану?

Якщо у функціональних програмах немає змінних, то функціональні програми не можуть змінювати стан. Як ми можемо очікувати, що програма буде корисною, якщо вона не може змінювати стан?

Відповідь полягає в тому, що функціональні програми обчислюють новий стан зі старого стану, не змінюючи старий стан. Якщо це звучить заплутано, то наступний приклад має все прояснити:

```
State system(State s) {
return isFinal(s) ? s : system(s);
}
```

Ви можете запустити system у деякому початковому стані, і вона буде послідовно переводити system зі стану в стан, поки не буде досягнутий кінцевий стан. system не змінює змінну стану. Натомість, на кожній ітерації зі старого стану створюється новий стан.

Якщо ми вимкнемо TCO і дозволимо стеку зростати з кожним рекурсивним викликом, то стек буде містити всі попередні стани без змін. Більше того, система функціонує як істинна функція в математичному сенсі. Якщо ви викликаєте system зі state1, вона щоразу повертатиме state2.

Якщо уважно подивитися на функціональну версію sumFirstTenSquares, то можна побачити, що вона використовує саме такий підхід до зміни стану. Тут немає ні змінних, ні внутрішнього стану.

Замість цього алгоритм рухається від початкового стану до кінцевого, по одній зміні стану за раз.

Звичайно, наша системна функція не здатна реагувати на будь-які вхідні дані. Вона просто починається з деякого початкового стану, а потім виконується до завершення. Але за допомогою простої модифікації ми можемо створити «функціональну» програму, що досить добре реагує на вхідні події:

```
State system(State state, Event event) {
return done(state) ? state : system(state, getEvent());
}
```

Тепер, обчислений наступний стан системи є функцією поточного стану та вхідної події. І вуаля — ми створили дуже традиційну машину станів (state machine), яка може реагувати на події в реальному часі.

Зверніть увагу на лапки, які я щойно поставив до слова «функціональну». Це тому, що getEvent не є прозорою до посилань. Кожного разу, коли ви її викликаєте, ви отримуватимете різні результати. Отже, ви не можете замінити виклик його значенням, що повертається. Чи означає це, що наша програма насправді «не-функціональна»?

Строго кажучи, будь-яка програма, що отримує вхідні дані у такий спосіб, не може бути суто функціональною. Але це не книжка про суто функціональні програми. Це книжка про функціональне програмування. Стиль програми, наведеної вище, є «функціональним», навіть якщо вхідні дані не є чистими; і саме цей стиль нас тут цікавить.

Пропоную вашій увазі просту маленьку скінченну машину станів, яка написана на C і є «функціональною». Це перевірений часом приклад турнікета в метро. Трохи порозважайтеся з нею.

```
#include ‹stdio.h›
typedef enum {locked, unlocked, done} State;
```

```
typedef enum {coin, pass, quit} Event;
void lock() {
printf("Locking.\n");
}
void unlock() {
printf("Unlocking.\n");
}
void thankyou() {
printf("Thanking.\n");
}
void alarm() {
printf("Alarming.\n");
}
Event getEvent() {
while (1) {
int c = getchar();
switch (c) {
case 'c': return coin;
case 'p': return pass;
case 'q': return quit;
}
}
}
State turnstileFSM(State s, Event e) {
switch (s) {
case locked:
switch (e) {
case coin:
unlock();
return unlocked;
case pass:
```

```
alarm();
return locked;
case quit:
return done;
}
case unlocked:
switch (e) {
case coin:
thankyou();
return unlocked;
case pass:
lock();
return locked;
case quit:
return done;
}
case done:
return done;
}
}
State turnstileSystem(State s){
return (s == done) ? 0 : turnstileSystem( turnstileFSM(s, getEvent()));
}
int main(int ac, char **av){
turnstileSystem(locked);
return 0;
}
```

Майте на увазі, що C не використовує TCO, тому стек буде рости, поки не вичерпається — хоча у цьому випадку може знадобитися досить багато операцій.

Незмінність

Усе це означає, що функціональні програми не містять змінних. Ніщо у функціональній програмі не змінює стану. Зміни стану передаються від одного виклику рекурсивної функції до іншого, не змінюючи жодного з попередніх станів. Якщо попередні стани непотрібні, TCO може оптимізувати їх; але вони все ще існуватимуть, незмінні, десь у минулому кадрі стеку.

Якщо у функціональній програмі немає змінних, то всі значення, які ми називаємо, є константами. Після ініціалізації ці константи ніколи не зникають і ніколи не змінюються. По суті, вся історія кожної із цих констант залишається недоторканою, незмінною і непорушною.

2 Постійні дані

Про шахрайство

Наші комп’ютери в певному сенсі є скінченними машинами Тюрінга; вони не засновані на лямбда-обчисленні. Теза Черча—Тюрінга говорить нам, що машини Тюрінга і лямбда-числення є еквівалентними формами; але це не означає, що ви можете легко перейти від однієї до іншої. Функціональна програма — це програма, яка виглядає як лямбда-числення, але реалізована на скінченній машині Тюрінга. І ця реалізація потребує певного шахраювання.

Першим обманом, який ми побачили, був TCO. Ми відмахнулися від нього, аргументуючи це прагматичними міркуваннями. Зрештою, якщо нам ніколи не знадобляться всі ці історичні кадри зі стеком, навіщо їх зберігати? Але це все одно шахрайство. «Під капотом» наша реалізація змінювала значення існуючих змінних. Із погляду машини Тюрінга, всі наші уявні константи насправді були змінними.

Ми можемо продовжувати штовхати цей чит-код угору. Цей чудовий маленький алгоритм Sieve працює повністю в конструкторі, тож це лише ініціалізація! І як ми вже дізналися, ініціалізація — це не присвоювання. Тож той факт, що ця програма має змінні «під капотом», нічим не відрізняється від TCO. Зрештою, результат усе одно функціональний.

Це весело! Ми можемо продовжувати просувати цей обман вгору. Ми можемо робити це, поки він не опиниться за межами нашої комп'ютерної машини Тюрінга. І тоді ми зможемо сказати собі: «Кожна програма, що працює у цьому комп'ютері, є функціональною, тому що вона завжди видає однакові результати, коли отримує однакові вхідні дані. Не важливо, що входи і виходи включають кожен біт у пам'яті комп'ютера. Не зважайте. Це те, що треба».

Звісно, якщо ми будемо дотримуватися такої точки зору, то немає сенсу вивчати функціональне програмування, хіба не так? Тож відступимо від цього шахрайства найвищого рівня і продовжимо відкидати чіти вниз доти, доки просто не зможемо уникнути їх.

Не існує розумного способу уникнути TCO. У нас немає нескінченного стека. Ми не хочемо, щоби наші функціональні програми марно витрачали гігабайти простору в стеку, поки не виникне збій. Отже, TCO — це практично неминучий обман.

Виготовлення копій

А як щодо алгоритму Sieve? Чи можемо ми знизити рівень шахрайства? Чи можемо написати цей алгоритм так, аби він не використовував операторів присвоювання?

Проблема, звісно, у всіх цих циклах. Нам потрібно перетворити їх на рекурсивні функції, щоби позбутися операторів присвоювання. Нам також потрібно щось зробити із двома масивами. Ми не

можемо змінювати елементи в існуючих масивах, чи не так? Бо це зробить такі масиви змінними. Отже, нам доведеться створювати їхні копії щоразу, коли нам знадобиться змінити елемент:

```
package sieve;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class Sieve {
static List‹Integer› primesUpTo(int upTo) {
return getPrimes(
computeSieve(
makeSieve(Math.max(upTo, 1)), 0),
new ArrayList‹›(), 0);
}
private static boolean[] makeSieve(int upTo) {
boolean[] sieve = new boolean[upTo + 1];
Arrays.fill(sieve, false);
sieve[0] = sieve[1] = true;
return sieve;
}
private static boolean[] computeSieve(boolean[] sieve, int n) {
if (n >= sieve.length)
return sieve;
else if (!sieve[n])
return computeSieve(markMultiples(sieve, n, 2), n + 1);
else return computeSieve(sieve, n + 1);
}
private static boolean[] markMultiples(boolean[] sieve, int prime, int
m) {
int multiple = prime * m;
if (multiple >= sieve.length)
```

```
    return sieve;
    else {
    var markedSieve = Arrays.copyOf(sieve, sieve.length);
    markedSieve[multiple] = true;
    return markMultiples (markedSieve, prime, m + 1);
    }
    }
    public static List‹Integer› getPrimes(boolean[] sieve, List‹Integer› primes, int n) {
    if (n>=sieve.length)
    return primes;
    else if (!sieve[n]) {
    var newPrimes = new ArrayList‹›(primes);
    newPrimes.add(n);
    return getPrimes(sieve, newPrimes, n+1);
    } else {
    return getPrimes(sieve, primes, n+1);
    }
    }
    }
```

Це не дуже гарно, чи не так? Однак це досить функціонально. Ви можете поскаржитися на призначення у makeSieve, і я згоден, що це трохи шахрайство, але це виглядає досить близько до ініціалізації, щоби задовольнити мене.

Отже, всі важливі операції присвоювання було вилучено. Всі іменовані сутності є константами, а стек (якщо його не видалено TCO) містить історію кожного виклику кожної рекурсивної функції.

Але якою ціною? Кожного разу, коли будь-який із двох масивів змінюється, створюється новий масив, аби запобігти зміні попереднього. Об’єм пам’яті, який використовує цей алгоритм, може бути величезним.

Уявіть, що потрібно знайти всі прості числа до 100 000. Скільки масивів sieve буде створено? Скільки масивів primes?

А як щодо часу виконання? Копіювання всіх цих масивів знову і знову з'їдатиме жахливу кількість циклів.

Чи не є це ціною функціонального програмування? Чи повинні ми жити з такою величезною витратою пам'яті та часу?

Структурний розподіл

На щастя, ні. Виявляється, існують структури даних, що поводяться дуже схоже на масиви, але при цьому ефективно зберігають історію своїх минулих станів. Ці структури даних є n-арними деревами. Що більше n, то ефективніше вони працюють. Але для простоти я обиратиму n із 2 — тобто бінарні дерева — для наступних прикладів.

Припустимо, що ми хочемо представити простий масив цілих чисел від 1 до 8. Бінарне дерево, що досягає цієї мети, показано на рис. 2.1.

Рис. 2.1. Бінарне дерево, що представляє масив цілих чисел [1..8]

Якщо ви подивитеся на листя і проігноруєте гілки, то побачите, що листя утворює масив. Гілки просто надають спосіб переходу до кожного листка у певному порядку. Цей порядок і є індексом масиву!

• • •