

ЗАДАЧИ ЗА ЗАДЪЛЖИТЕЛНА САМОПОДГОТОВКА

ПО

Увод в програмирането

email: kalin@fmi.uni-sofia.bg

4 декември 2018 г.

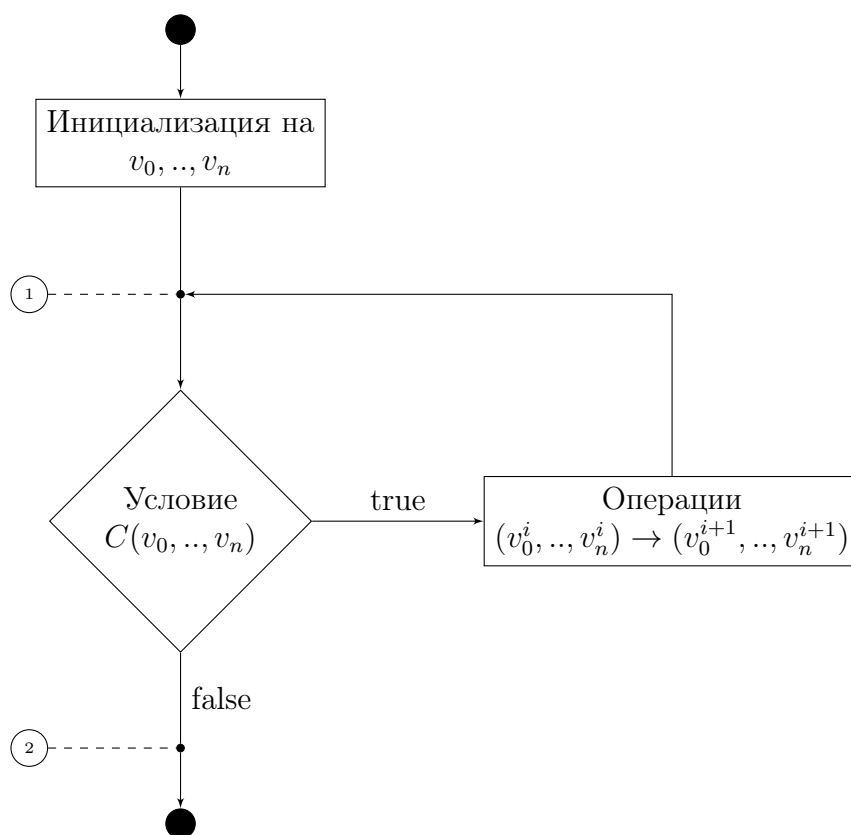
1 Инварианта на цикъл

Изложението в тази секция е силно опростена версия на метода на Флойд (Floyd) за верификация на блок схеми, описан в монументалната му статия [2]. Редица идеи и задачи са взимствани от книгата [1] на А. Скоскова и С. Николова “Теория на програмите в задачи”. За по-подробно изложение и допълнителни примери и задачи препоръчваме тази книга. Изложението по-долу е нарочно опростено, като на места е жертвана прецизността му.

Инварианта на цикъл

Нека е дадена програма или част от програма, която се състои от единствен **while** цикъл. Нека имаме някакъв набор от работни променливи v_1, \dots, v_n , които се инициализират непосредствено преди **while** цикъла. Условието на цикъла зависи изцяло от работните променливи, а тялото на цикъла използва или променя само тях. Приемаме, че програмата не проивежда странични ефекти и не зависи от такива.

Тоест, за програмата знаем, че: (1) не извършва входни и изходни операции, (2) поведението ѝ зависи изцяло от началните стойности на работните променливи v_1, \dots, v_n и (3) не модифицира никакви други променливи, освен работните. Такава програма можем да изобразим чрез блок схемата на Фигура 1.



Фигура 1: Блок схема на част от програма с **while** цикъл

На фигурата, $C(v_1, \dots, v_n)$ е някакъв логически израз, зависещ само от v_1, \dots, v_n . Да допуснем, че сме “паузирали” изпълнението на програмата в точката, обозначена с “1”, точно преди да започне i -тото поред ($i \geq 0$) изпълнение на тялото на цикъла, т.е. непосредствено преди да бъдат изпълнени операторите в него за i -ти път. В този момент, n -торката работни променливи (v_1, \dots, v_n) имат някакви конкретни стойности. Да ги обозначим с (v_0^i, \dots, v_n^i) . След изпълнение на всички оператори в тялото на цикъла, работните променливи получават някакви нови стойности. Това са точно стойностите $(v_0^{i+1}, \dots, v_n^{i+1})$, които работните променливи ще имат в началото на $i+1$ -вата итерация. Това е изобразено на фигурата чрез прехода $(v_0^i, \dots, v_n^i) \rightarrow (v_0^{i+1}, \dots, v_n^{i+1})$.

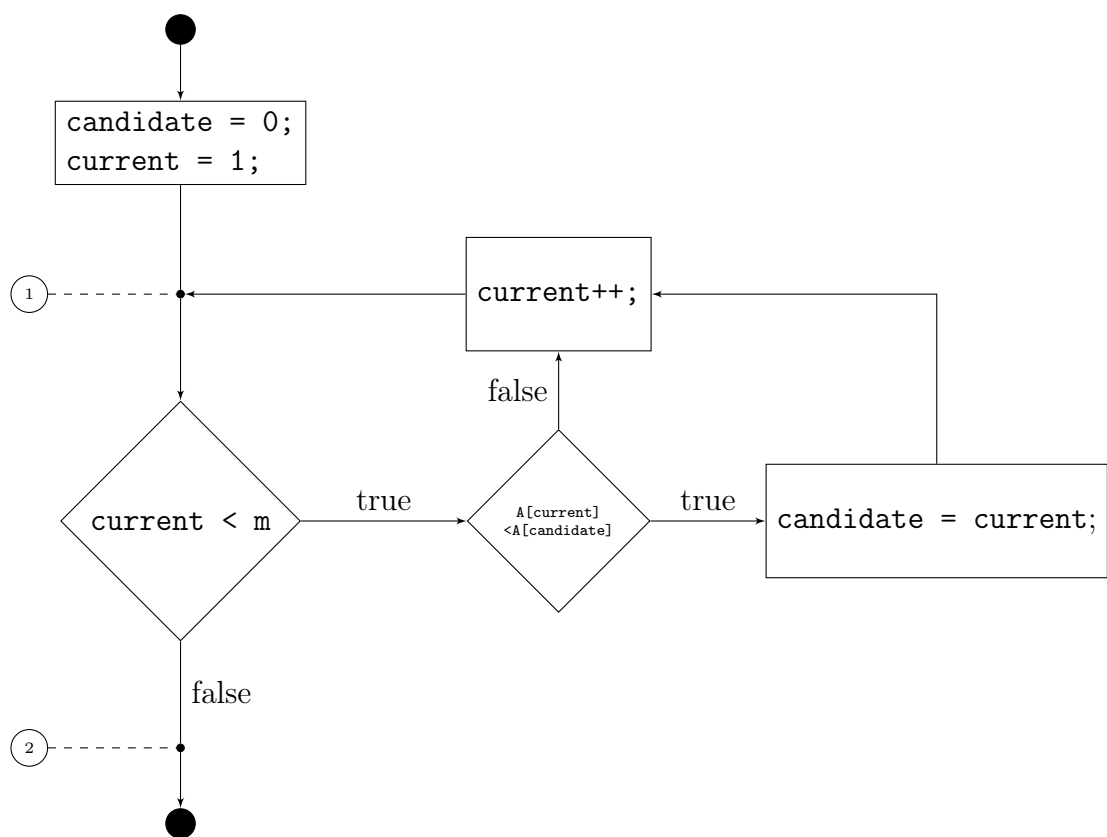
Нека приемем, че при някакво конкретно изпълнение на програмата, при конкретни дадени начални стойности на работните променливи (v_0^0, \dots, v_n^0) , тялото на цикъла ще бъде изпълнено точно $k \geq 0$ пъти, след което условието $C(v_1, \dots, v_n)$ на цикъла ще се наруши и той ще приключи. Изпълнението на програмата достига точката, обозначена с “2”. По този начин се получава редицата от n -торки $(v_0^0, \dots, v_n^0), \dots, (v_0^k, \dots, v_n^k)$, като за всички нейни членове $0 \leq i < k$, освен за последния, знаем че е вярно $C(v_0^i, \dots, v_n^i)$, а за последния, k -ти член, е вярно обратното — $\neg C(v_0^k, \dots, v_n^k)$.

Освен свойството $C(v_0^i, \dots, v_n^i)$ (или неговото отрицание), което знаем за всички последователни стойности на работните променливи, можем да въведем още едно свойство $I(v_0^i, \dots, v_n^i)$, наречено “инварианта на цикъла”. За свойството I искаме да е вярно за всички възможни стойности на работните променливи, дори и за последния член на редицата им. От там идва името “инварианта”, т.е. факт, което е винаги верен.

Това свойство може да е произволно, например тавтологичната истина **true** изпълнява условието за инварианта, тъй като е вярна за всички членове на редицата на работните променливи. Инвариантата обаче е най-полезна, когато закодира “смисъла” на работните променливи, и ако от верността на $\neg C(v_0^k, \dots, v_n^k) \& \& I(v_0^k, \dots, v_n^k)$ можем да изведем нещо полезно за “крайният резултат” от изпълнението на цикъла, съдържащ се в стойностите (v_0^k, \dots, v_n^k) .

Верификация на програми

Понятието “Инварианта” ще илюстрираме чрез едно нейно приложение: “Верификация на програма”. Верификацията на програма е



Фигура 2: Блок схема на програмата за намиране на най-малък елемент в масив

доказателство, че при необходимите начални условия дадена програма изчислява стойност, удовлетворяваща някакво желано свойство. Като пример да разгледаме следната програма, за която ще се уверим, че намира най-малък елемент на едномерен масив A с $m > 0$ елемента $A[0], \dots, A[m - 1]$.

```
size_t candidate = 0, current = 1;
while (current < m)
{
    if (A[candidate] < a[current])
    {
        candidate = current;
    }
    current++;
}
```

Можем да приложим метода на ивариантата, за да докажем строго, че когато цикълът приключи, то елементът $A[\text{candidate}]$ е гарантирано по-малък или равен на всички останали елементи на масива. Като първа стъпка, за улеснение изобразяваме програмата като блок схемата на Фигура 2.

Работните променливи на програмата, освен масива A , са целочислените променливи **candidate** и **current**. Какъв е смисълът на тези променливи? Веднага се вижда, че **current** е просто брояч - служи за обхождане на елементите на масива последователно от $A[0]$ до $A[m - 1]$, като на всяка итерация от цикъла се разглежда стойността на $A[\text{current}]$.

Интуитивно се вижда, че **candidate** е намерения най-малък елемент на масива до текущия момент от обхождането. Тоест можем да се надяваме, че ако сме разгледали първите i елемента на масива, то правилно сме определили, че най-малкият от тях е $A[\text{current}]$.

Тези размисления може да запишем чрез инвариантата:

$$I ::= A[\text{candidate}] = \min(A[0], \dots, A[\text{current} - 1]).$$

Тази инварианта очевидно е вярна в началото на изпълнението на програмата, когато $\text{current} = 1$, а $\text{candidate} = 0$. Как да се уверим, че инвариантата е валидна за всички стойности, през които преминават работните променливи?

Нека допуснем, че инвариантата е вярна за някаква стъпка i от изпълнението на цикъла. Тоест, допускаме $I(\text{candidate}^i, \text{current}^i)$, или $A[\text{candidate}^i] = \min(A[0], \dots, A[\text{current}^i - 1])$.

На итерация $i + 1$ имаме $candidate^{i+1} = current^i$ и $current^{i+1} = current^i + 1$, и следователно трябва да покажем, че $A[candidate^{i+1}] = \min(A[0], \dots, A[current^{i+1} - 1])$.

След навлизане в тялото на цикъла, имаме две възможности:

1) $A[current^i] \geq A[candidate^i]$. От допускането следва, че добавянето на $A[current^i]$ към $\min(A[0], \dots, A[current^i - 1])$ не променя стойността на минимума, или $\min(A[0], \dots, A[current^i - 1]) = \min(A[0], \dots, A[current^i - 1], A[current^i])$. От тук директно се вижда, че инвариантата е вярна за итерация $i + 1$.

2) $A[current^i] < A[candidate^i]$. Тъй като по допускане $A[candidate^i] = \min(A[0], \dots, A[current^i - 1])$, от тук следва, че $A[current^i] < \min(A[0], \dots, A[current^i - 1])$, следователно

$$A[current^i] = \min(A[0], \dots, A[current^i - 1], A[current^i]).$$

Но след присвояването имаме $candidate^{i+1} = current^i$, от където $A[candidate^{i+1}] = \min(A[0], \dots, A[current^i])$. Но $current^i = current^{i+1} - 1$, от където получаваме верността на I за итерация $i + 1$: $A[candidate^{i+1}] = \min(A[0], \dots, A[current^{i+1} - 1])$.

Доказателството протича по индукция. Уверяваме се, че инвариантата е вярна за цялата редица от стойности на **candidate** и **current**.

Какво се случва в края на цикъла, когато имаме $\neg(current < m)$? Тъй като числата са цели, а **current** се увеличава само с единица, можем да заключим, че $current = m$. Замествайки това равенство в инвариантата получаваме твърдението

$$A[candidate] = \min(A[0], \dots, A[m - 1]).$$

Тоест, намерили сме минималния елемент на масива.

1.1. Да се докаже строго, че за следната функция е вярно $pow(x, y) = x^y$:

```
a) unsigned int pow (unsigned int x, unsigned int y)
{
    unsigned int p = 1, i = 0;
    while (i < y)
    {
        p *= x;
        i++;
    }
    return p;
}
```

```

6) unsigned int pow (unsigned int x, unsigned int y)
{
    unsigned int p = 1;
    while (y > 0)
    {
        p *= x;
        y--;
    }
    return p;
}

```

```

B) unsigned int pow (unsigned int x, unsigned int y)
{
    unsigned int z = x, t = y, p = 1;
    while (t > 0)
    {
        if (t%2 == 0)
        {
            z *= z;
            t /= 2;
        }else{
            t = t - 1;
            p *= z;
        }
    }
    return p;
}

```

1.2. Да се докаже строго, че за следната функция е вярно $\text{sqrt}(n) = \lfloor \sqrt{n} \rfloor$.

```

unsigned int sqrt (unsigned int n)
{
    unsigned int x = 0, y = 1, s = 1;
    while (s <= n)
    {
        x++;
        y += 2;
        s += y;
    }
    return x;
}

```

- 1.3. Да се напише програма, която проверява дали дадени два масива A и B с еднакъв брой елементи m са еднакви, т.е. съдържат същите елементи в същия ред. Да се докаже строго, че програмата работи правилно.
- 1.4. Да се докаже, че следната функция връща истина тогава и само тогава, когато масивът A с n елемента съдържа елемента x .

```
bool member (int A[], size_t n, int x)
{
    size_t i = 0;
    while (i < n && A[i] != x)
    {
        i++;
    }
    return i < n;
}
```

- 1.5. Да се напише програма, която проверява дали даден масив A с m елемента е сортиран, т.е. дали елементите му са наредени в нарастващ ред. Да се докаже строго, че програмата работи правилно.

Съществуват редица съвременни методи за верификация на програми. Пълната версия на метода, използван по-горе, се нарича “логика на Флойд-Хоар” (Floyd–Hoare logic) и е само един представител на този клас методи.

Също така, в компютърните науки има направление, наречено “Синтез на програми” (Program synthesis). При синтеза на програми се решава обратната задача: по спецификация на входа и изхода да се генерира програма, която удовлетворява спецификацията.

Препоръчваме на любознателния читател да се запознае с логиката на Floyd–Hoare и методите за синтез на програми.

Литература

- [1] Александра Соскова, Стела Николова, “Теория на програмите в задачи”, Софтех, 2003
- [2] R. W. Floyd. “Assigning meanings to programs.” Proceedings of the American Mathematical Society, Symposia on Applied Mathematics. Vol. 19, pp. 19–31. 1967.