# Poglavje 7

# Dokazovanje pravilnosti programov

Pri razvoju programov je osnovna zahteva, da je program pravilen, tj. da za vse pravilne vhodne podatke v končnem času dobimo pravilen rezultat. Pravilnost programa lahko preverjamo na več načinov:

- z branjem programa in intuitivnim razumevanjem, kaj program dela,
- s testiranjem programa na izbrani množici testnih podatkov,
- s formalnim dokazom pravilnosti programa.

Prvi dve metodi sta skoraj vedno nepopolni. Z branjem in razumevanjem programa kaj hitro lahko spregledamo napako na isti način, kot jo je programer. Pri testiranju programov lahko dokažemo samo napačnost delovanja programa, pravilnost pa lahko dokažemo samo za tiste vhodne podatke, na katerih smo program dejansko izvajali.

Tretja metoda je najzahtevnejša, saj zahteva natančno analizo programa in formalen opis pomembnih relacij med podatkovnimi objekti med izvajanjem programa. Ker so dokazi pravilnosti programa tipično daljši od samega programa, je seveda možno, da se pojavijo napake tudi v dokazu. Zato je navidezno, formalno dokazovanje pravilnosti bolj samo sebi namen kot res koristno orodje. Poleg tega zaradi formalne zahtevnosti to metodo v praksi bolj redko uporabljamo.

Vseeno je formalno dokazovanje pravilnosti koristno:

- omogoča boljše razumevanje, kaj pravilnost programa dejansko pomeni, ter razjasni težavnost preverjanja pravilnosti programa;
- za določene algoritme, ki jim intuitivno ni moč zaupati, lahko formalen dokaz enkrat za vselej pokaže pravilnost algoritma;

- pri kritičnih delih kode, kjer bi napačno delovanje povzročilo katastrofalne posledice, kot npr. krmilnik telefonske centrale, s formalnim dokazom pravilnosti zagotovimo zanesljivost programske opreme.

V naslednjih razdelkih je opisano dokazovanje pravilnosti preprostih programov (Manna, 1974), ki uporabljajo

- zaporedje stavkov,
- prireditveni stavek,
- izbiro **if-else**,
- zanko **while**

## 7.1 Pravila izpeljevanja pogojev

Program definiramo kot preslikavo:

$$f : X \longrightarrow Z$$

ki preslika vhodne podatke $< x_1, ..., x_n > \in X$ v izhodne podatke
$< z_1, ..., z_m > \in Z$. Pri tem morajo vhodni podatki izpolnjevati *začetni pogoj*:

$$\phi(x_1, ..., x_n)$$

Izhodni podatki pa morajo izpolnjevati zaključni pogoj:

$$\psi(z_1, ..., z_m, x_1, ..., x_n)$$

pri čemer predpostavimo, da se vrednosti vhodnih podatkov med izvajanjem programa ne spreminjajo. Pravimo, da je program:

**parcialno pravilen,** če v primeru, da se za vhodne podatke, ki izpolnjujejo začetni pogoj $\phi$, ustavi, izhodni podatki izpolnjujejo zaključni pogoj $\psi$;

**totalno pravilen,** če je parcialno pravilen, in če se za vse vhodne podatke, ki izpolnjujejo začetni pogoj $\phi$, po končnem številu korakov ustavi.

Pri dokazovanju pravilnosti programa iz pogojev $P$, ki veljajo pred izvrševanjem stavka, izpeljujemo pogoje $Q$, ki veljajo po izvršitvi stavka:

```
// P(Y)
Stavek;
// Q(Y)
```

Opišimo še pravila, ki se pri tem uporabljajo. Argumenti pogojev so (pomembne) spremenljivke. Pravila so:

**Prireditev:**

```
// P(izraz)
y = izraz;
// P(y)
```

**Izbira:**

```
//P(y)
if (Pogoj(y))
  // P(y) & Pogoj(y)
    ...;
  else
  // P(y) & !Pogoj(y)
    ...;
```

**Zanka:**

```
// P1(y)
 while (Pogoj(y)) {
  // za i−to izvajanje zanke: Pi(y) & Pogoj(y)
    S;
 } // while
// Pk(y) & !Pogoj(y)
```

Pri čemer velja

```
// Pi(y) & Pogoj(y)
 S
// P_(i+1)(y)
```

in je pogoj $P_k(y)$ odvisen od števila izvajanj zanke.

**Zaporedje:**

*// P0(y)*
{ S1; S2; ...; Sk }
*// Pk(y)*

pri čemer velja

*// P_(i−1)(y)*
Si;
*// Pi(y)*

**Združitev več poti izvajanja:**

**if** (...)
{ ... }
*// P1(y)*
**else**
{...}
*// P2(y)*
; *// P1(y) or P2(y)*

## 7.2 Parcialna pravilnost programa

Program je parcialno pravilen, če v primeru, da se za vhodne podatke, ki izpolnjujejo začetni pogoj $\phi$, ustavi, izhodni podatki izpolnjujejo zaključni pogoj $\psi$.

Za to, da izpeljemo zaključni pogoj $\psi$ iz začetnega pogoja $\phi$, uporabljamo pravila, opisana v prejšnjem razdelku. Edino problematično pravilo je pri zanki, saj je izstopni pogoj $P_k(y)$ odvisen od števila izvajanj zanke, ki pa ga vnaprej ne poznamo.

Temu problemu se izognemo tako, da definiramo *zančno invarianto*, tj. pogoj $I(y)$, ki je vsebovan v vseh pogojih $P_i$, torej je resničen pred začetkom izvajanja zanke, pred vsakim ponovnim izvajanjem zanke in po zaključku izvajanja zanke:

*// I(y)*
**while** (Pogoj(y)) {
*// I(y) & Pogoj(y)*
S;
*// I(y)*
} *// while*
*// I(y) & !Pogoj(y)*

Tak pogoj mora biti zadosti "močan", da omogoči izpeljavo zaključnega pogoja $\psi$ (trivialen pogoj, ki je v zanki vedno resničen, je pogoj $true$, ki pa očitno ne omogoča izpeljave zaključnega pogoja).

Ko poznamo primerno zančno invarianto, je dokaz parcialne pravilnosti programa rutinsko delo, ki slepo sledi pravilom iz prejšnjega razdelka. Dejanska teža vsakega dokaza parcialne pravilnosti programa je torej v definiciji ustrezne zančne invariante.

## 7.3 Totalna pravilnost programa

Program je totalno pravilen, če je parcialno pravilen, in če se za vse vhodne podatke, ki izpolnjujejo začetni pogoj $\phi$, po končnem številu korakov ustavi. Zopet so pri preverjanju ustavljivosti programa problematične samo zanke.

Za dokaz ustavljivosti zanke je potrebno definirati:

- zančno spremenljivko $l$,
- *dobro utemeljeno množico* (delno urejeno množico brez neskončnih padajočih zaporedij) $D$ (ponavadi vzamemo kar množico naravnih števil),
- zančno invarianto $l \in D$,

ter dokazati, da je zančna invarianta $l \in D$ resnična pred začetkom izvajanja zanke in pred vsakim ponovnim izvajanjem zanke ter da se vrednost spremenljivke $l$ po vsaki izvršitvi zanke zmanjša. Ker je zaloga vrednosti $D$ spremenljivke $l$ dobro utemeljena množica, ki je za vsako padajoče zaporedje navzdol omejena, nam ti pogoji zagotavljajo, da se bo zanka po končnem številu korakov iztekla. Z $l$ označimo vrednost zančne spremenljivke pred izvajanjem telesa zanke, z $ll$ pa označimo novo vrednost zančne invariante po zaključku enega izvajanja telesa zanke:

*// l pripada D*
**while** (Pogoj(y)) {
  *// l pripada D & Pogoj(y)*
    S;
  *// ll pripada D & (ll < l)*
 } *// while*