KORREKTHET

BEVIS AV ATT ETT PROGRAM UPPFYLLER SIN SPECIFIKATION

SPECIFIKATION AV ETT (DEL-)PROGRAMS FUNKTION:

INGÅNGSVILLKOR **PRE** SOM MÅSTE VARA UPPFYLLT DÅ PROGRAMMET STARTAS. (FÖRUTSÄTTNING)

UTGÅNGSVILLKOR **POST** SOM MÅSTE VARA UPPFYLLT DÅ PROGRAMMET AVSLUTAS.

PROGRAMBEVIS:

MATEMATISKT/LOGISKT BEVIS AV ATT OM **PRE** ÄR SANT OCH PROGRAMMET EXEKVERAS TILL SLUT SÅ KOMMER SEDAN **POST** ATT VARA SANT.

PRE {PROGRAM} **POST**

DET ÄR OMÖJLIGT ATT SKRIVA ETT PROGRAM SOM BEVISAR ATT ETT PROGRAM ÄR KORREKT!

GRADER AV KORREKTHET

PARTIELL KORREKTHET

FÖR VARJE INDATA SOM UPPFYLLER **PRE** SÅ KOMMER PROGRAMMET ANTINGEN ATT AVSLUTAS MED **POST** UPPFYLLT ELLER ATT ALDRIG AVSLUTAS.

TOTAL KORREKTHET

FÖR VARJE INDATA SOM UPPFYLLER **PRE** SÅ KOMMER PROGRAMMET ATT AVSLUTAS OCH **POST** ATT VARA UPPFYLLT.

DET ÄR OFTAST ENKLAST ATT FÖRST VISA PARTIELL KORREKTHET OCH SEDAN VISA ATT PROGRAMMET ALLTID AVSLUTAS FÖR KORREKTA INDATA

SLINGOR OCH REKURSIONER BRYTS $\Rightarrow$ PROGRAMMET AVSLUTAS

STUDERA ETT HELTALSUTTRYCK SOM MINSKAR (ÖKAR) I VARJE VARV I SLINGAN/REKURSIVT ANROP OCH SOM HAR EN UNDRE (ÖVRE) ABSOLUT GRÄNS!

TIPS FÖR PROGRAMBEVISNING

- SPECIFICERA VARJE PROCEDURS (ÖNSKADE) BETEENDE MED INGÅNGS- OCH UTGÅNGSVILLKOR.

  VISA PRE {PROCEDURKOD} POST

- SPECIFICERA SLINGOR MED INVARIANTER. INVARIANTEN SKA VARA SANN PRECIS INNAN SLINGAN BÖRJAR OCH EFTER VARJE VARV I SLINGAN.

- SPECIFICERA VARJE VIKTIGT LÄGE MED EN FÖRSÄKRAN (ASSERTION)

EXEMPEL:

```
PRE A
KOD 1
while .. { INV B
  KOD 2 }
KOD 3
ASSERT C
KOD 4
POST D
```

BEVISA FÖLJANDE:

A {KOD 1} B

B {KOD 2} B

B {KOD 3} C

C {KOD 4} D

- GRÄV INTE NER DEJ I DETALJER!

Loopinvariant

En loopinvariant är ett enkelt men ändå kraftfullt verktyg för att bättre förstå slingor (while- och for-satser). En väl vald loopinvariant är till nytta både när man skriver programmet, när man testar det och när man modifierar det. En loopinvariant fungerar dessutom som dokumentation och kan utgöra grunden för ett korrekthetsbevis.

Exempel 1: summa

Ett enkelt exempel får illustrera hur loopinvarianter fungerar.

```
/**
 * Returnerar summan 1 + 2 + ... + n, där n >= 1.
 */
public long sum(int n) {
    long sum = 0;
    int i = 1;
    while (i <= n) {
        // Invariant: sum = 1 + 2 + ... + (i - 1)
        sum += i;
        i++;
    }
    return sum;
}
```

En invariant är ett påstående om variablerna i ett program som är sant varje gång programmet passerar invarianten. I det här avsnittet kommer vi speciellt att titta på loopinvarianter: invarianter som står i början (eller slutet) av en slinga. En bra loopinvariant har följande tre egenskaper.

- **Initiering:** Loopinvarianten måste vara sann innan vi exekverar slingan första gången. Detta stämmer i exemplet ovan eftersom `sum = 0` och `i = 1` i detta fall. (Den tomma summan är noll.)
- **Uppdatering:** Om invarianten är sann i en iteration av slingan så måste den vara sann även i nästa iteration. Om vi i exemplet ovan antar att loopinvarianten är korrekt i den i:te iterationen så kommer den även att vara sann även i nästa iteration eftersom vi adderar `i` till summan och därefter ökar `i` med 1.
- **Avslutning:** När loopen avslutas så ska loopinvarianten säga något användbart som hjälper oss att förstå algoritmen. I exempelt ovan säger loopinvarianten att `sum = 1 + 2 + ... + n` just innan slingan avslutas, dvs precis det som behövs för att metoden ska vara korrekt.

De tre stegen ovan är i själva verket ett induktionsbevis som visar att `sum = 1 + 2 + ... + n` när programmet lämnar slingan.

Exempel 2: max

I nästa exempel använder vi en loopinvariant redan när vi designar algoritmen. Vi börjar med följande kodskelett.

```
/**
 * Returnerar det maximala värdet i vektorn v.
 */
public int max(int[] v) {
    int max = ... ;
```

```
    for (int i = 0; i < v.length; i++) {
        // Invariant: max = max(v[0], v[1], ..., v[i-1])
        ...
    }
    return max;
}
```

Invarianten verkar vara väl vald: den uppfyller ju den tredje egenskapen (avslutning) eftersom den säger att variabeln `max = max(v[0], v[1], ..., v[v.length-1])` när slingan avslutas.

Den första egenskapen (initiering) kan vi försöka uppfylla genom att att tilldela `max` ett lämpligt startvärde. Det visar sig svårt eftersom det är oklart hur man definierar maxvärdet av en tom vektor. Vi har hittat en potientiell bugg i programmet redan innan vi har skrivit koden! Det finns flera lösningar. Jag väljer att ändra metodens kontrakt:

```
/**
 * Returnerar det maximala värdet i vektorn v, där v.length > 0.
 *
 * @throws IllegalArgumentException om v.length = 0.
 */
public int max(int[] v) {
    if (v.length == 0)
        throw new IllegalArgumentException("v.length = 0");

    int max = ... ;
    for (int i = 1; i < v.length; i++) {
        // Invariant: max = max(v[0], v[1], ..., v[i-1])
        ...
    }
    return max;
}
```

Nu är det lätt all lösa problemet. Om vi låter `max = v[0]` och startar slingan med `i = 1` så är loopinvarianten uppfylld första gången vi kommer in i slingan.

Nu återstår bara att skriva koden inuti slingan. *När vi gör det så kan vi utgå från att loopinvarianten gäller i början av slingan.* Egenskap två (uppdatering) säger nämligen: *Om* invarianten är sann i en iteration av slingan så måste den vara sann även i nästa iteration. Den färdiga metoden ser då ut så här.

```
/**
 * Returnerar det maximala värdet i vektorn v, där v.length > 0.
 *
 * @throws IllegalArgumentException om v.length = 0.
 */
public int max(int[] v) {
    if (v.length == 0)
        throw new IllegalArgumentException("v.length = 0");

    int max = v[0] ;
    for (int i = 1; i < v.length; i++) {
        // Invariant: max = max(v[0], v[1], ..., v[i-1])
        if (v[i] > max) {
            max = v[i];
        }
    }
    return max;
}
```

Exempel 3: insättningssortering

```
/**
 * Kastar om ordningen på elementen i vektorn v så att
 * v[0] <= v[1] <= ... <= v[v.length-1].
 */
public void sort(int[] v) {
    for (int j = 1; j < v.length; j++) {
        // Invariant: delvektorn v[0..j-1] består av
        // samma element som ursprungsvektorn v[0..j-1]
        // men i sorterad ordning.
        int key = v[j];
        int i = j - 1;
        while (i >= 0 && v[i] > key) {
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```

För att uppnå egenskap tre (avslutning) så behöver vi en loopinvariant som säger att elementen är sorterade och att det dessutom är samma element som i ursprungsvektorn.

Vi börjar med att kontrollera den första egenskapen (initiering). Vi kan anta att vektorn innehåller minst två element eftersom for-loopen annars inte exkeveras. (Det behövs inte heller eftersom en vektor med 0 eller 1 element alltid är sorterad.) Innan slingan exekveras säger loopinvarianten att "delvektorn v[0..1-1] består av samma element som ursprungsvektorn v[0..1-1] men i sorterad ordning." Detta påstående är sant.

För att verifiera egenskap två (uppdatering) måste vi titta lite närmre på koden. Vi antar att loopinvarianten håller (dvs `v[0..j-1]` är sorterad) och behöver bara kontrollera att koden sätter in elementet `v[j]` på rätt plats i delvektorn `v[0..j]`. Detta sker genom att man flyttar alla element `v[j-1]`, `v[j-2]`,... som är större än `v[j]` ett steg till höger och därefter sätter in `v[j]` på sin rätta plats.

Om man vill så kan man naturligtvis använda en loopinvariant även för att kontrollera att koden i while-loopen är korrekt. Det kan vara en bra övning.

Stefan Nilsson [https://plus.google.com/+StefanNilsson/about?rel=author]