

## Unterabschnitt 4.4.2

### Verifikation von Prozeduren

©2010

TU Kaiserslautern

780

### Verifikation von Prozeduren

Wir betrachten nur die Prinzipien der Verifikation nicht-rekursiver Prozeduren und verzichten auf den Nachweis von Terminierung.

Um zu zeigen, dass eine nicht-rekursive Prozedur ihre Spezifikation aus Vor- und Nachbedingung erfüllt, muss man zeigen, dass ihr Rumpf, also eine Anweisung, die Vor- und Nachbedingung erfüllt.

Die *Verifikationsaufgabe* zu

- einer Vorbedingung **P**
- einer Anweisung **A** und
- einer Nachbedingung **Q**

bekommt damit folgende Form:

©2010

TU Kaiserslautern

781

## Verifikation von Prozeduren (2)

Zeige, dass die Ausführung von **A**

- beginnend in Vorzuständen, in denen **P** gilt,
- bei Terminierung in Nachzuständen endet, in denen **Q** gilt.

Abkürzend schreibt man dafür üblicherweise:

{ **P** } **A** { **Q** } (Hoare-Triple)

Zur formalen Verifikation von Hoare-Tripeln benutzt man eine *Hoare-Logik* oder den WP-Kalkül (siehe GdP).

Wir betrachten hier eine verwandte Verifikationstechnik, die auf Zusicherungen basiert und auf Floyd zurückgeht.

©2010

TU Kaiserslautern

782

## Syntaktische Einschränkungen

Um die Darstellung zu vereinfachen machen wir für Prozedurrümpfe folgende Einschränkungen:

- keine Prozeduraufrufe
- while-Anweisungen
- Fallunterscheidungen/if-Anweisungen
- Zuweisungen (nicht in Bedingungen)
- seiteneffektfreie Ausdrücke
- alle Variablendeklarationen am Anfang
- Rückgabe-Anweisung nur am Ende des Rumpfs und zwar `return result;`

©2010

TU Kaiserslautern

783

## Beispiel:

```
int sqrt( int x ) {
    int result , c , s ;
    c = 0 ;
    s = 1 ;
    while ( s <= x ) {
        c = c+1 ;
        s = s + ( 2*c+1 ) ;
    }
    result = c ;
    return result ;
}
```

vgl. Prozedur f von Folie 773

©2010

TU Kaiserslautern

784

## Resultierende Verifikationsaufgabe:

```
assert x >= 0 ; // ist vom Aufrufer sicherzustellen
c = 0 ;
s = 1 ;
while ( s <= x ) {
    c = c+1 ;
    s = s + ( 2*c+1 ) ;
}
result = c ;
assert result*result <= x && x < ( result+1)*( result+1 ) ;
return result ;
```

©2010

TU Kaiserslautern

785

## Verifikationstechnik:

Annotiere das Programm so mit Zusicherungen, dass:

- vor und hinter jeder Anweisung mindestens eine Zusicherung steht und
- die im Folgenden erläuterten Regeln erfüllt sind.

### Bemerkungen:

- Eine solche Verifikation nennt man auch einen **Annotationsbeweis** (engl. **proof outline**).
- Die Zusicherungsanweisung ist eine Java-Anweisung, die die Bedingung für die aktuellen Werte zur Laufzeit (!) prüft.

©2010

TU Kaiserslautern

786

## Zuweisungsregel:

```
assert P[(E)/x] ;
x = E ;
assert P ;
```

wobei  $P[(E)/x]$  den Ausdruck bezeichnet, den man erhält, wenn man in  $P$  alle Vorkommen von  $x$  durch  $(E)$  ersetzt. Überflüssige Klammern lassen wir im Folgenden weg.

### Bemerkungen:

Die korrekte Zusicherung für den Vorzustand lässt sich aus der Zusicherung für den Nachzustand berechnen.

©2010

TU Kaiserslautern

787

## Beispiele: (Zuweisungsregel)

Korrekt gemäß Zuweisungsregel:

```
assert c*c <= x  &&  x < (c+1)*(c+1) ;
result = c;
assert result*result <= x  &&  x < (result+1)*(result+1);
```

Korrekt gemäß Zuweisungsregel:

```
assert c*c <= x  &&  s+(2*c+1) == (c+1)*(c+1) ;
s = s + (2*c + 1);
assert c*c <= x  &&  s == (c+1)*(c+1) ;
```

Inkorrekt gemäß Zuweisungsregel:

```
assert x == 8 ;
x = y;
assert x == 8 ;
```

©2010

TU Kaiserslautern

788

## Abschwächungsregel:

Zwei Zusicherungen dürfen aufeinander folgen

```
assert P;
assert Q;
```

wenn die Zusicherung  $P$  die Zusicherung  $Q$  impliziert, d.h. wenn in allen Zuständen, in denen  $P$  gilt, auch  $Q$  gilt.

©2010

TU Kaiserslautern

789

## Beispiele: (Abschwächungsregel)

Korrekte Abschwächung:

```
assert s > x  &&  c*c <= x  &&  s == (c+1)*(c+1) ;
assert c*c <= x  &&  x < (c+1)*(c+1) ;
```

Inkorrekte Abschwächung:

```
assert x <= y ;
assert x < y-1 ;
```

©2010

TU Kaiserslautern

790

## Schleifenregel:

Schleifen müssen wie folgt annotiert sein:

```
assert INV;
while( b ) {
  assert b && INV;
  S
}
assert INV;
assert !b && INV;
```

wobei INV ein geeigneter Ausdruck/eine geeignete Formel ist.

©2010

TU Kaiserslautern

791

## Begriffsklärung: (Schleifeninvariante)

Sei  $W = \text{while}(b)\{S\}$  eine Schleife mit seiteneffektfreier Bedingung  $b$ .

Eine **Schleifeninvariante**  $INV$  zur Schleife  $W$  ist ein boolescher Ausdruck/eine logische Formel mit der Eigenschaft:

```
{ b && INV } S { INV }
```

### Bemerkungen:

- Die zentrale Schwierigkeit bei der Verifikation von nicht-rekursiven prozeduralen Programmen ist das Entwickeln geeigneter Schleifeninvarianten.
- Schleifeninvarianten sind auch ein hervorragendes Dokumentationsmittel, da sie die algorithmische Idee ausdrückt, die der Schleife zugrunde liegt.

©2010

TU Kaiserslautern

792

## Beispiele: (Schleifenregel)

$INV = \text{def } true$  ist korrekt gemäß Schleifenregel, aber wenig hilfreich:

```
assert true;
while ( c<x ) {
  assert c<x && true;
  assert true;
  r = r + y;
  assert true;
  c = c + 1;
  assert true;
}
assert c >= x && true;
```

©2010

TU Kaiserslautern

793

## Beispiele: (Schleifenregel) (2)

Korrekt gemäß Schleifenregel und hinreichend stark:

```
assert c <= x && r == c*y;
while ( c<x ) {
  assert c<x && c <= x && r == c*y;
  assert c+1<=x && r+y==(c+1)*y;
  r = r + y;
  assert c+1<=x && r == (c+1)*y ;
  c = c + 1;
  assert c<=x && r == c*y ;
}
assert c>=x && c<=x && r == c*y ;
assert c==x && r == c*y ;
assert r==x*y ;
```

©2010

TU Kaiserslautern

794

## Verzweigungsregel:

Fallunterscheidungen müssen wie folgt annotiert sein:

```
assert P;
if ( b ) {
  assert b && P;
  S1
  assert Q;
} else {
  assert !b && P;
  S2
  assert Q;
}
assert Q;
```

wobei  $b$  die Bedingung ist und  $P$  und  $Q$  geeignete boolesche Ausdrücke/logische Formeln.

©2010

TU Kaiserslautern

795

## Beispiel: (Annotationsbeweis)

```

assert x >= 0; // ist vom Aufrufer sicherzustellen
assert 0*x <= x && 1==(0+1)*(0+1);
c = 0;
assert c*c<=x && 1==(c+1)*(c+1);
s = 1;
assert c*c<=x && s==(c+1)*(c+1);
while( s <= x ) {
  assert s<=x && c*c<=x && s==(c+1)*(c+1);
  assert (c+1)*(c+1)<=x && s==(c+1)*(c+1);
  c = c+1;
  assert c*c<=x && s== c*c;
  assert c*c<=x && s+(2*c+1) == (c+1)*(c+1);
  s = s + (2*c+1);
  assert c*c<=x && s==(c+1)*(c+1);
}
assert s>x && c*c<=x && s==(c+1)*(c+1);

```

©2010

TU Kaiserslautern

796

## Beispiel: (Annotationsbeweis) (2)

```

assert s>x && c*c<=x && s==(c+1)*(c+1);
assert c*c<=x && x<(c+1)*(c+1);
result = c;
assert result*result <= x && x < (result+1)*(result+1);
return result;

```

### Bemerkung:

Wie schon bei Spezifikationen reicht es auch in vielen Beweisaufgaben nicht aus, nur boolsche Ausdrücke über den sichtbaren Variablen und Parametern zu verwenden.

©2010

TU Kaiserslautern

797

## Beispiel: (Verikation unter Verwendung mathematischer Funktionen)

Sei *Fac* die mathematische Fakultätsfunktion auf den natürlichen Zahlen mit den Eigenschaften:

$$\text{Fac}(0) = 1 \quad [\text{E1}]$$

$$\text{für alle } n > 0 \text{ gilt: } \text{Fac}(n) = \text{Fac}(n-1) * n \quad [\text{E2}]$$

©2010

TU Kaiserslautern

798

## Beispiel: (Verikation unter Verwendung mathematischer Funktionen) (2)

Wir betrachten folgende Verifikationsaufgabe:

```

int i, r, result;
assert 0 <= x && x <= 12;
i = 0;
r = 1;
while( i < x )
  // invariant 0 <= i && i <= x && r==Fac(i)
  {
    i = i + 1;
    r = r * i;
  }
result = r;
assert result == Fac(x);
return result;

```

©2010

TU Kaiserslautern

799

## Beispiel: (Verikation unter Verwendung mathematischer Funktionen) (3)

Schleifeninvariante:

```
0 <= i && i <= x && r == Fac(i)

int i, r, result;
assert 0<x && x <=12;
// E1: Fac(0)==1 gilt in jedem Zustand
assert 0<=0 && 0<=x && 1==Fac(0) ;
i = 0;
assert 0<=i && i <=x && 1==Fac(i) ;
r = 1;
assert 0<=i && i <=x && r==Fac(i) ;
while( i < x ) {
  assert i<x && 0<=i && i <=x && r==Fac(i) ;
  assert 0<i+1 && i+1<=x && r*(i+1)==Fac((i+1)-1)*(i+1);
```

©2010

TU Kaiserslautern

800

## Beispiel: (Verikation unter Verwendung mathematischer Funktionen) (4)

```
assert 0<i+1 && i+1<=x && r*(i+1)==Fac((i+1)-1)*(i+1);
i = i + 1;
assert 0<i && i <=x && r*i==Fac(i-1)*i;
// E2: wg. i>0 gilt Fac(i)== Fac(i-1)*i
assert 0<=i && i <=x && r*i==Fac(i) ;
r = r * i;
assert 0<=i && i <=x && r==Fac(i) ;
}
assert i>=x && 0<=i && i <=x && r==Fac(i) ;
assert i==x && r==Fac(i) ;
assert r==Fac(x) ;
result = r;
assert result==Fac(x) ;
return result;
```

©2010

TU Kaiserslautern

801

## Erweiterte Anwendung der Verifikationstechnik:

Für die Anwendung der Verifikationstechnik auf allgemeinere Programme betrachten wir:

1. Vereinfachung von Programmen
2. Erweiterung der Ausdruckssprache

Wir erläutern diese Aspekte an einem Beispiel.

©2010

TU Kaiserslautern

802

## Beispiel: (Formale/informelle Verikation)

Als realistisches Verifikationsbeispiel betrachten wir die Prozedur `partition` unserer quicksort-Implementierung (Folie ??):

```
/*@ requires 0 <= ug < og < f.length ;
   @ modifies f[*] ;
   @ let PK = f[og].key ;
   @ ensures
     ug <= \result <= og
     && f[\result].key == PK
     && (\forallall i; ug <=i <\result; f[i].key<=PK)
     && (\forallall i; \result <i<og; f[i].key>=PK)
   @*/
int partition( DataSet[] f, int ug, int og){
  ... }
```

©2010

TU Kaiserslautern

803

## 1. Vereinfachung der Programmsyntax:

Ersetze so weit möglich alle Sprachkonstrukte, die nicht von der Verifikationstechnik unterstützt werden, durch äquivalente Formulierungen.

Ersetze nicht-rekursive Prozeduren ggf. durch ihren Rumpf.

Am Beispiel der Implementierung von `partition`:

- Ersetzen aller Postinkrement- und –dekrement-Anweisungen
- Einführen der Variable `result`
- Ersetzen der Variableninitialisierungen
- Ersetzen der Prozedur `swap` durch ihren Rumpf

Aus der Fassung von Folie ?? ergibt sich damit die Fassung der Folien 805f.

©2010

TU Kaiserslautern

804

## Neue Version von `partition`

```
int partition( DataSet[] f, int ug, int og){
    DataSet dtmp;
    int left, pk, right, result;
    left = ug;
    pk = f[og].key;
    right = og-1;
    while( left <= right ) {
        while( f[left].key < pk ) {
            left = left+1;
        }
        while( left <= right && f[right].key >= pk ) {
            right = right-1;
        }
    }
}
```

©2010

TU Kaiserslautern

805

## Neue Version von `partition` (2)

```
if( left <= right ) {
    dtmp = f[left];
    f[left] = f[right];
    f[right] = dtmp;
    left = left+1;
    right = right-1;
} else { ; }
}
dtmp = f[left];
f[left] = f[og];
f[og] = dtmp;
return result;
}
```

©2010

TU Kaiserslautern

806

## Bemerkungen:

- Bei der Anpassung können natürlich Fehler passieren.
- Nicht alle Sprachkonstrukte kann man ohne weiteres Ersetzen; relativ einfach sind:
  - ▶ Vermeiden von `break` und `return` innerhalb des Rumpfes, ...
  - ▶ Standardisieren von Schleifen
  - ▶ Ausdrücke vereinfachen
- Datenstrukturen und Rekursion brauchen gesonderte Verifikationstechniken

©2010

TU Kaiserslautern

807

## 2. Erweiterung der Ausdruckssprache:

Die Spezifikation von `partition` demonstriert drei Erweiterungen:

- Verwendung des Quantors `\ forall`
- Verwendung von Feldern und Verbunden
- Verwendung der logischen Variablen `PK`. Nötig, weil:
  - ▶ die Komponente `f[og]` im Laufe des Partitionierens ggf. verändert wird und
  - ▶ der Wert des Pivotelements im Nachzustand ansonsten durch Parameter nicht ausgedrückt werden kann.

Bei der Verifikation werden wir diese Erweiterungen entsprechend unserem allgemeinen Programmverständnis behandeln.

### Bemerkung:

Für alle diese Erweiterungen gibt es formale Beweisregeln.

©2010

TU Kaiserslautern

808

## Beweisaufgabe für `partition`

Finde Invariante `INV` und Annotation `ANNOT`, so dass die folgenden Teile 1-4 verifiziert werden können:

```

assert 0 <= ug && ug < og
      && og < f.length && PK == f[og].key ;
left = ug;
pk = f[og].key;
right = og-1;
assert INV ;

```

} Teil 1

## Beweisaufgabe für `partition` (2)

```

assert left <= right && INV
while( f[left].key < pk ) {
  left = left+1;
}
while( left <= right && f[right].key >= pk ) {
  right = right -1;
}
assert ANNOT

```

} Teil 2

©2010

TU Kaiserslautern

810

## Beweisaufgabe für `partition` (3)

```

assert ANNOT
if( left <= right ) {
  ...
} else {
  ;
}
assert INV

```

} Teil 3

©2010

TU Kaiserslautern

809

TU Kaiserslautern

811



## Beweisaufgabe für partition (4)

```

assert !(left <= right) && INV ;
dtmp = f[left];
f[left] = f[og];
f[og] = dtmp;
result = left;
assert ug <= result && result <= og
    && f[result].key == PK
    && (!forall i; ug <= i < result; f[i].key <= PK)
    && (!forall i; result < i <= og; f[i].key >= PK)

```

} Teil 4

Beweis in fünf Schritten:

- A. Finden der Invarianten INV
- B–E. Beweisen der Teile 1–4.

©2010

TU Kaiserslautern

812

## A. Finden der Invarianten

Algorithmische Idee (vgl. Folie 727ff):

Indexzähler `left`, `right` laufen von links bzw. rechts bis

```
f[left].key >= pivot.key && f[right].key < pivot.key
```

Es gilt:

```
\forall all i in [ug, left - 1] : f[i].key < pivot.key
\forall all i in [right + 1, og] : pivot.key <= f[i].key
```

Desweiteren dokumentieren wir: `PK == pk == f[og].key`

Insgesamt:

```
PK == pk == f[og].key && ug <= left <= right + 1 <= og
&& (!forall i; ug <= i < left; f[i].key < pk)
&& (!forall i; right < i < og; f[i].key >= pk)
```

©2010

TU Kaiserslautern

813

## B. Verifizieren von Teil 1:

```

assert 0 <= ug < og < f.length && PK == f[og].key ;
assert PK == f[og].key && ug <= og <= og
    && (!forall i; ug <= i < og; f[i].key < f[og].key) ;
left = ug;
assert PK == f[og].key == f[og].key && ug <= left <= og
    && (!forall i; ug <= i < left; f[i].key < f[og].key) ;
pk = f[og].key;
assert PK == pk == f[og].key && ug <= left <= og
    && (!forall i; ug <= i < left; f[i].key < pk) ;

```

©2010

TU Kaiserslautern

814

## B. Verifizieren von Teil 1: (2)

```

assert PK == pk == f[og].key && ug <= left <= og
    && (!forall i; ug <= i < left; f[i].key < pk) ;
assert PK == pk == f[og].key && ug <= left <= og - 1 + 1 <= og
    && (!forall i; ug <= i < left; f[i].key < pk)
    && (!forall i; og - 1 < i < og; f[i].key >= pk) ;
right = og - 1;
assert PK == pk == f[og].key && ug <= left <= right + 1 <= og
    && (!forall i; ug <= i < left; f[i].key < pk)
    && (!forall i; right < i < og; f[i].key >= pk) ;

```

©2010

TU Kaiserslautern

815

## C. Verifizieren von Teil 2:

Als Invariante für die erste innere Schleife verwenden wir **INV**.  
Zu zeigen bleibt dann für den Rumpf (von hinten lesen!):

```

assert f[left].key < pk
    && PK==pk==f[og].key && ug <= left <= right+1 <= og
    && (\forall forall i; ug <= i < left; f[i].key < pk )
    && (\forall forall i; right < i < og; f[i].key >= pk );

assert f[left].key < pk
    && PK==pk==f[og].key && ug <= left <= right+1 <= og
    && (\forall forall i; ug <= i < left; f[i].key < pk )
    && (\forall forall i; right < i < og; f[i].key >= pk );
    && f[right+1].key >= pk && left != right+1
    && left <= right

```

©2010

TU Kaiserslautern

816

## C. Verifizieren von Teil 2: (2)

```

assert f[left].key < pk
    && PK==pk==f[og].key && ug <= left <= right+1 <= og
    && (\forall forall i; ug <= i < left; f[i].key < pk )
    && (\forall forall i; right < i < og; f[i].key >= pk );
    && left <= right

assert PK==pk==f[og].key && ug <= left+1<= right+1 <= og
    && (\forall forall i; ug <= i < left+1; f[i].key < pk)
    && (\forall forall i; right < i < og; f[i].key >= pk );

left = left+1;

assert PK==pk==f[og].key && ug <= left <= right+1 <= og
    && (\forall forall i; ug <= i < left; f[i].key < pk )
    && (\forall forall i; right < i < og; f[i].key >=pk );

```

©2010

TU Kaiserslautern

817

## C. Verifizieren von Teil 2: (3)

Als Invariante für die zweite innere Schleife nehmen wir

```
f[left].key >= pk && INV
```

Zu zeigen bleibt dann für den Rumpf:

```

assert left <= right && f[right].key >= pk
    && f[left].key >= pk
    && PK==pk==f[og].key && ug <= left <= right+1 <= og
    && (\forall forall i; ug <= i < left; f[i].key < pk )
    && (\forall forall i; right < i < og; f[i].key >=pk );

```

©2010

TU Kaiserslautern

818

## C. Verifizieren von Teil 2: (4)

```

assert f[left].key >= pk
    && PK==pk==f[og].key && ug <= left <= right <= og
    && (\forall forall i; ug <= i < left; f[i].key < pk )
    && (\forall forall i; right-1< i < og; f[i].key >= pk );

right = right-1;

assert f[left].key >= pk
    && PK==pk==f[og].key && ug <= left <= right+1 <= og
    && (\forall forall i; ug <= i < left; f[i].key < pk )
    && (\forall forall i; right < i < og; f[i].key >= pk );

```

©2010

TU Kaiserslautern

819

## C. Verifizieren von Teil 2: (5)

**ANNOT** bietet sich damit die Zusage nach der Schleife an:

```
( left > right || f[right].key < pk )
&& f[left].key >= pk
&& INV
```

©2010

TU Kaiserslautern

820

## D. Verifizieren von Teil 3:

Zunächst der then-Zweig (von hinten lesen):

```
assert left <= right
&& ( left > right || f[right].key < pk )
&& f[left].key >= pk
&& INV

assert left <= right
&& f[right].key < pk && f[left].key >= pk
&& PK==pk==f[og].key && ug <= left <= right+1 <= og
&& (\forall forall i; ug <= i < left; f[i].key < pk )
&& (\forall forall i; right < i < og; f[i].key >= pk )

assert PK==pk==f[og].key && ug <= left < right < og
&& (\forall forall i; ug <= i < left; f[i].key < pk )
&& f[right].key < pk && f[left].key >= pk
&& (\forall forall i; right < i < og; f[i].key >= pk );
```

©2010

TU Kaiserslautern

821

## D. Verifizieren von Teil 3: (2)

```
assert PK==pk==f[og].key && ug <= left < right < og
&& (\forall forall i; ug <= i < left; f[i].key < pk )
&& f[right].key < pk && f[left].key >= pk
&& (\forall forall i; right < i < og; f[i].key >= pk );

dtmp = f[left];
f[left] = f[right];
f[right] = dtmp;

assert PK==pk==f[og].key && ug <= left < right < og
&& (\forall forall i; ug <= i < left; f[i].key < pk )
&& f[left].key < pk && f[right].key >= pk
&& (\forall forall i; right < i < og; f[i].key >= pk );
```

©2010

TU Kaiserslautern

822

## D. Verifizieren von Teil 3: (3)

```
assert PK==pk==f[og].key && ug <= left < right < og
&& (\forall forall i; ug <= i < left; f[i].key < pk )
&& f[left].key < pk && f[right].key >= pk
&& (\forall forall i; right < i < og; f[i].key >= pk );

assert PK==pk==f[og].key && ug <= left+1 <= right-1 <= og
&& (\forall forall i; ug <= i < left+1; f[i].key < pk )
&& (\forall forall i; right-1 < i < og; f[i].key >= pk );

left = left+1;
right = right-1;

assert PK==pk==f[og].key && ug <= left <= right+1 <= og
&& (\forall forall i; ug <= i < left; f[i].key < pk )
&& (\forall forall i; right < i < og; f[i].key >= pk );
```

©2010

TU Kaiserslautern

823

## D. Verifizieren von Teil 3: (4)

Für den else-Zweig müssen wir zeigen, dass

```
left > right && ANNOT
```

die Invariante **INV** impliziert, und das ist offensichtlich der Fall, da **ANNOT** den Ausdruck **INV** als Konjunkt enthält.

©2010

TU Kaiserslautern

824

## E. Verifizieren von Teil 4:

```
assert left > right
&& PK==pk==f[og].key && ug <= left <= right+1 <= og
&& (\forall forall i; ug <= i < left; f[i].key < pk )
&& (\forall forall i; right < i < og; f[i].key >= pk );

assert ug <= left <= og && f[og].key == PK
&& left == right+1;
&& (\forall forall i; ug <= i < left; f[i].key < PK )
&& (\forall forall i; right < i <= og; f[i].key >= PK );

assert ug <= left <= og && f[og].key == PK
&& (\forall forall i; ug <= i < left; f[i].key < PK )
&& (\forall forall i; left <= i <= og; f[i].key >= PK );
```

©2010

TU Kaiserslautern

825

## E. Verifizieren von Teil 4: (2)

```
assert ug <= left <= og && f[og].key == PK
&& (\forall forall i; ug <= i < left; f[i].key <= PK )
&& (\forall forall i; left < i < og; f[i].key >= PK )
&& f[left].key >= PK ;

dtmp = f[left];

assert ug <= left <= og && f[og].key == PK
&& (\forall forall i; ug <= i < left; f[i].key <= PK )
&& (\forall forall i; left < i < og; f[i].key >= PK )
&& dtmp.key >= PK ;

f[left] = f[og];

assert ug <= left <= og && f[left].key == PK
&& (\forall forall i; ug <= i < left; f[i].key <= PK )
&& (\forall forall i; left < i < og; f[i].key >= PK )
&& dtmp.key >= PK ;
```

©2010

TU Kaiserslautern

826

## E. Verifizieren von Teil 4: (3)

```
assert ug <= left <= og && f[left].key == PK
&& (\forall forall i; ug <= i < left; f[i].key <= PK )
&& (\forall forall i; left < i < og; f[i].key >= PK )
&& dtmp.key >= PK ;

f[og] = dtmp;

assert ug <= left <= og && f[left].key == PK
&& (\forall forall i; ug <= i < left; f[i].key <= PK )
&& (\forall forall i; left < i < og; f[i].key >= PK )
&& f[og].key >= PK ;
```

©2010

TU Kaiserslautern

827

## E. Verifizieren von Teil 4: (4)

```
assert ug <= left <= og && f[left].key == PK
&& (\forallall i; ug <= i < left; f[i].key <= PK )
&& (\forallall i; left < i < og; f[i].key >= PK )
&& f[og].key >= PK ;

assert ug <= left <= og && f[left].key == PK
&& (\forallall i; ug <= i < left; f[i].key <= PK )
&& (\forallall i; left <= i <= og; f[i].key >= PK );

result = left;

assert ug <= result <= og && f[result].key == PK
&& (\forallall i; ug <= i < result; f[i].key <= PK )
&& (\forallall i; result < i <= og; f[i].key >= PK );
```