Vorlesung 25

16.09.2024

Korrektheitsbeweise

Wiederholung:

- strukturelle Induktion
- 3 Schritte für Korrektheit
    - (I) Invariante finden
        a. Inv(0) gilt vor erstem Durchlauf
        b. (vor i-tem Durchlauf:) Inv(i-1) → Inv(i) (← nach i-tem Durchlauf)
- (II) Schleife terminiert
- (III) Nach letztem Durchlauf m: In(m) AND NOT(while-Bed.) ⇒ Korrektheit

Korrektheitsbeweis Selectionsort Funktional

```
// Vor: list.length>0
// Erg: kleinstes Elemente der Liste ist geliefert
def minimum[A:Ordering](list:List[A]):A=
    val ord = summon[Ordering[A]]
    import ord.mkOrderingOps
    list match
        case Nil => throw Exception("Minimum of empty list")
        case List(x) => x
        case (x::xs) => x min (minimum(xs))

// Vor: keine
// Erg: eine Liste mit allen Elementen aus list in aufsteigen
def selectionSortA:Ordering:List[A]=
```

```
    val ord = summon[Ordering[A]]
    import ord.mkOrderingOps
    def removeFirst[A](list:List[A], q:A):List[A]=
        list match
            case Nil => Nil
            case (x::xs) if (x==q) => xs
            case (x::xs) => x::removeFirst(xs,q)
        list match
            case Nil => Nil
            case _ => val m = minimum(list)
                val rest = removeFirst(list,m)
                m::selectionSort(rest)
```

- Um die Korrektheit von Selectionsort zu beweisen muss zunächst die Korrektheit von `minimum` und `removeFirst` bewiesen werden → Tutorium & Übung
- Anschließend wird eine Hauptrekursion mit struktureller Induktion über `list` durchgeführt:
    - I.A. `|list| = 0` bzw. `list = Nil`
      Geben Nil zurück. Nil ist sortiert und hat alle Elemente der Ursprungsliste.
    - I.V. Für n beliebig, aber fest, gibt selectionSort für alle Listen der Länge <= n eine
      sortierte Liste mit genau den Elementen der Eingabeliste aus
    - I.S. `n -> n+1` bzw. `xs -> x::xs`
      - list hat mindestens ein Element
      - damit gibt
      `minimum` ein Element m mit `m <= y für alle y aus list` zurück
      - Also kann m in einer sortierten Reihenfolge vorne stehen
      -
      `removeFirst(list,m)` entfernt das erste Vorkommen von m aus list
      -
      `m::removeFirst(list,m`) enthält also genau die Elemente aus list
      - nach I.V. enthält dann
      `m::selectionSort(removeFirst(list,m))`
      alle Elemente aufsteigend sortiert

Korrektheitsbeweis Selectionsort Imperativ

```
# selectionsort(List[int]):None
#Eff: Die Elemente in list sind aufsteigend sortiert
#Erg: keins
def selectionsort(xs):
    n = len(xs)
    # Eff: keiner
    # Erg: Der index des ersten minimalen Elements der Liste
    def min(i):
        index = i+1
        m = i
        while (index < n):
            if (xs[index]<xs[m]):
                m = index
            index = index +1
        return m
    index = 0
    while(index < n):
        minindex = min(index)
        xs[index], xs[minindex] = xs[minindex], xs[index]
        index = index +1
```

- `minimum` wird in Tutorium besprochen
- (I) Invariante finden und für grundlegende Fälle als geltend Zeigen
  - Inv(k):
    a. Die Elemente an Index 0,...,k-1 von xs sind aufsteigend sortiert
    b. Alle Elemente an Index 0,...,k-1 sind kleiner oder gleich aller Elemente an Index k,...,n-1.
    c. Index = k
    d. Menge der Elemente in xs ist gleich wie bei der Eingabe
  - `Inv(0)`: vor dem ersten Durchlauf:

    a. & b. 0,...,-1 ist leere Sequenz, damit gelten a) + b) direkt

    c. index = 0, d. trivial
  - Angenommen, `Inv(i-1)` gilt vor i-tem Durchlauf, also c) ⇒ `index = i-1`

Da `minimum` korrekt ist, ist `minindex` der Index einers kleinsten Elements an Indizex `i-1,...,n-1`
Wegen b) von
`Inv(i-1)` ist `list[minindex]` ein kleinstes Element, welches größer/gleich der Elemente an Index 0,...,i-2 ist. ()
Nach dem Tausch an Position index ist Liste 0,...,i-1 dann wegen ()
und `Inv(i-1)` a) aufsteigend sortiert. `(Inv(i)` a)) Und wegen (*) gilt auch `Inv(i)` b).
Wegen index = index + 1 als letzte Zeile und Inv(i-1) c) gilt auch Inv(i) c).

- (II) Nach n Durchläufen gilt `Inv(n)` c) also ist `index = n >= n` und die Schleife endet.
- (III) Nach der Schleife gilt `Inv(n)`. es gilt a), b), c) und dazu `index >= n` (Negation der while-Bedingung). Korrektheit folgt direkt aus `Inv(n)` a).

Invariantenbeweis für Partion und Quicksort

Invariante → Muss in Loops gelten = Korrektheit

Partition

Die Invariante wurde definiert als:

INV(k):

a) alle Elemente an Index start+1 ... l sind kleiner/gleich pivot
b) alle Elemente an Index l+1 ... start+k sind größer pivot
c) die Menge der Elemente and start...end-1 ist gleich wie bei der Eingabe
d) i = start + k + 1

gegeben:

```
def partition(xs,start, end):
    pivot = xs[start]
    l = start
    i = start + 1
    while(i<end):
        if (xs[i]<=pivot):
            l = l + 1
            xs[i], xs[l] = xs[l], xs[i]
        i = i+1
```

```
    xs[start], xs[l] = xs[l], xs[start]
    return l
```

Wenn Inv(a-1) vor Durchlauf a gilt, dann gilt Inv(a) nach Durchlauf a.

1. Inv(a) zeigen

   Inv(a) d) → i = start + a

   ▼ Fall 1: `xs[i]≤ pivot`

   `xs[i], xs[l+1] = xs[l+1], xs[i]` wird ausgeführt.

   a. Nach Tausch ist `xs[l+1] ≤ pivot`, nach `l = l +1` gilt Inv(a) a) wieder.

   b. Nach Inv(a-1) b) 9st vor Tausch `xs[l+1] > pivot`

      Also nach dem Tausch `xs[i] > pivot` mit `i = start + a` folgt `xs[start +a ]>pivot`, also gilt Inv(a) b)

   c. Da die Elemente nur getauscht werden gilt Inv(a) c)

   d. Durch `i = i +1` ist `i = start + a -1 + 1 = start +a` → Inv(a) d) gilt

   ▼ Fall 2: `xs[i] > pivot`

   a. l wird hier nicht verändert, also gilt auch a) weiterhin

   b. Nach Voraussetzung des Falls ist xs[start + a]> pivot woraus folgt, dass b) weiterhin für Inv(a) gilt

   c. Da nichts an der Liste geändert wird, gilt Inv(a) c) trivialerweise weiterhin

   d. Durch i = i +1 istt i = start + a -1 + 1 = start +a → Inv(a) d) gilt

2. Terminierungsbedingung

   In jedem Durchlauf wird i inkrementiert, und nach `end - start -1` Durchläufen ist `i = end` und die Schleife terminiert

3. Es gilt zu zeigen, dass nach dem letzten Schleifendurchlauf gelten die Bedingungen weiterhin und, dass die Terminierung der Schleife erreicht ist.

   Nach Schleife gilt Inv(end-start-1) also gelten:

   a. `xs[j]≤ pivot` für `start + 1 ≤ j ≤ l` und

b. `xs[j] > pivot` für `l + 1 ≤ j < end` und

c. kein Element dazu oder verschwunden

d. Nach finalem Tausch: `xs[j] ≤ pivot für start ≤ j < l`, `xs[j] >pivot für l+1 ≤ j < end`, `xs[l] = pivot`

Korrektheitsbeweis für Quicksort basierend auf dem Partition beweis

→ Wir brauchen keine Invariante da wir Induktion über die Anzahl der zu sortierenden Elemente machen & in-place sortieren, deswegen ist `n = end - start`

```
# quicksort(List[int]):List[int]
#Vor: keine
#Eff: Die Elemente in list sind aufsteigend sortiert
#Erg: keins
def quicksort(xs):
    def quicksort_help(start, end):
        if (end-start) <=1:
            return
        split = partition(xs, start, end)
        quicksort_help(start, split)
        quicksort_help(split+1, end)
    quicksort_help(0,len(xs))
```

I.A.:

- n = 0, → die leere Liste ist sortiert
- n = 1 → die einelementige Teilliste ist sortiert

I.V.:

Für ein beliebiges aber festes n sortiert quicksort_help in allen Aufrufen die `end-start ≤ n` korrekt

I.S.: n → n+1

(*) Nach dem Aufruf von `partition` ist nach dem obigen Beweis sind die Elemente in `xs[start:split]` alle `≤ xs[split]`, die Elemente in `xs[split+1:end]` alle `>xs[split]`

Da `split-1 < end` und `split +1 ≥ start` ist `(split -1) - start ≤ n` und `end - (split +1) ≤ n`

Also sind nach der I.V. bzw. (*) nach den rekursiven Aufrufen `xs[start:split]` und `xs[split+1:end]` aufsteigend sortiert

d

Dann ist auch `quicksort(xs) = quicksort_help(xs, 0, len(xs))` korrekt ■

Objektorientiertes Scala

Scala hat zwei Sorten von variablen, val und var. val kennen wir bereits, es kann kein neues Objekt zugewiesen werden. var hingegen kann ein neues Objekt zugewiesen werden.

```
def ifExample():Unit=
    var i: Int = 0
    i = i+1
    println(i)
```

Syntax von Schleifen in Scala

```
//alternative: Do-While loop, beingung wird am Ende der Schle
//dowhile(1) => 1 2 4 8, doWhile(20) => 20
def doWhile(i:Int):Unit =
    var num = i
    while
        println(num)
        num = num*2
        num < 10
    do()
```

For schleifen:

for variable ← collection do block, dabei ist collection der name einer Sammlung z.b. List, Array,... über die iteriert wird.

```
//for loop
//Iterieren über Zahlenbeeiche, end of rangeist in Scala inkl
//Weitere Varianten in der Scala Dokumentation

def forExample(list:List[Int]): Unit =
    for x <- list do
        println(x)
    println("For Range")
    for x <- 1 to 10 do
        println(x)
```

Listen in Scala sind unveränderbar, Zuweisung erfolgt mit runden klammern, alle Elemente müssen den gleichen Typ haben und der Zugriff auf einen Index i hat die Laufzeit O(i)

Listen in Python hingegen sind veränderbar, Zugriff erfolgt über [] eckige Klammern, unterschiedliche Elemente sind erlaubt und zugriff benötigt eine Laufzeit von O(1)

Arrays in Scala sind veränderbar, zugriff erfolgt über () und Zugriff auf Index i in O(1) Zeit. Die Größe des Arrays ist NICHT dynamisch und muss bei Initialisierung mit übergeben werden.

Das Verhalten von Arrays in Scala entspricht dem Array Verhalten vieler anderer Programmiersprachen wie C++, Java etc. hier ist eher Python ein Outlier.

Syntax zum erzeugen:

`val array1: Array[Type] = Array[Type](a0,…,an)` → Array mit den Elementen

`val array2: Array[Type] =new Array[Type](n)` → leeres Array mit n freien Plätzen

```
def arrayExample():Unit=
    var array:Array[Int] = Array[Int](1,2,3,4,5)
    for i <- 0 to array.length-1 do
        println(array(i))

    //schöner
    for x <- array do
        println(x)
```

```
    //schreiben
    val array2 = new Array[Int](4)
    var i = 0
    for a <- 2 to 5 do
        array(i) = a
        i = i +1
    for x <- array2 do
        println(x)
```

Achtung: Danke an den Unterschied zwischen val und var ! Einem val Array kann kein neues Array zugewiesen werden. Natürlich sind auch mehrdimensionale Arrays möglich

Bsp:

```
//Vorraussetzung: n >= 1
//Effekt: Keiner
//ergebnis:: ein Array von Arrays mit n Zeilen und n Spalten
def multTable(n:Int):Array[Array[Int]] =
    var table: Array[Array[Int]] = new Array[Array[Int]](n)
    for i <- 0 to table.length-1 do
        table(i) = new Array[Int](n)
    for i <- 0 to n-1 do
        for j <- 0 to n-1 do
            table(i)(j) = (i+1)*(j+1)
    table
```