Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Prof. Dr. W.-P. de Roever
Martin Steffen

Algorithmen & Datenstrukturen

Wintersemester 2002/2003 **Handout 2** November 2002

Handout 2: Rekursion und Iteration

Ausgabetermin: November 2002

Fangen wir erstmal mit der einfachsten Darstellung von *Quicksort* an, ohne Arrays und dem Rumgehühnere mit den Indizes (s. Abbildung 1). Die Funktion `qsort` nimmt die zu sortierende Liste `l`, wählt, falls sie nicht leer ist, das erste Element `x` als *Pivot.* Der Rest `tl` der Liste wird in zwei Hälften geteilt, in Elemente *kleiner* und eine Hälfte *größer oder gleich* als der Pivot. Diese Scheidung besorgt die Hilfsfunktion `filter`. Die Teile werden rekursiv sortiert und, zusammen mit dem Pivot, zu der sortierten Liste zusammengefügt. Die Korrektheit inklusive Terminierung des Algorithmus' liegt auf der Hand.

```
module QuicksortFun (E: ORDERED) : (SORT with type elem = E.t) =
    struct
      type elem = E.t

      let rec filter p l = match l with  (* auxiliary function *)
        []      -> []
      | x :: tl ->
          if   (p x)
          then  x::(filter p tl)
          else  filter p tl

      let rec sort (l : elem list) =
        match l with
          []  -> []
        | x :: tl  ->                                    (* x = pivot *)
            sort (filter (function y -> E.lt(y,x)) tl)
            @ [x] @
            sort (filter (function y -> E.geq(y,x)) tl)
    end
```

Abbildung 1: Quicksort

Als nächstes eine *Array-Implementierung* von Quicksort (s. Abbildung 2). Anders als die in der Vorlesung vorgestellte Lösung, die mittels *until*-Schleifen arbeitete, verwenden

wir hier *while*-Schleifen, ansonsten geht es genauso. Allerdings ist die Array-Lösung nicht identisch mit der aus Abbildung 1, indem in der Array-Variante die *Trennung* in obere und untere Hälfte nicht mit der Trennung aus dem Algorithmus von Abbildung 1 übereinstimmen muß, daß also der von `partition` zurückgegebene Indexwert `q` *nicht* mit dem Index des Pivotelementes indentisch sein muß. Funktioniert aber trotzdem, macht es nur schwieriger, drüber nachzudenken und die Korrektheit ist, sagen wir, nicht offensichtlich.

```
(* Imperative implementation of quicksort, using (integer) arrays *)

let qsort_a_r (a: int array) =
  let partition ((p: int), (r: int)) =
    (if (p < r) then
      (
      let x = a.(p)                                          (* Pivot      *)
      and i = ref p and j = ref r and h = ref 0
      in
      while (!i)<=(!j)
      do
        while a.(!i) < x  do i := !i + 1; done;
        while a.(!j) > x  do j := !j - 1; done;
        if (!i)<(!j)
        then
          (h := a.(!i); a.(!i) <- a.(!j); a.(!j) <- !h;   (* exchange *)
           i := !i+1; j:=!j-1)
        else (i:=!i+1)
      done;
      !j)                                                    (* return *)
    else p)
  in let rec qsort((p: int), (r: int)) =
    if (p < r)
    then
      (let q = ref 0
      in
      q := partition(p,r);
      qsort(p,!q);
      qsort(!q+1,r))
    else
      ()
  in qsort(0, Array.length a - 1);;
```

Abbildung 2: Array-Implementierung von Quicksort

Nun zur Aufgabe: wie wird man die Rekursion los? Die beiden aufeinanderfolgenden rekursiven Aufrufe von *qsort* machen Quicksort *nicht endrekursiv.* Will man derartige Fälle iterativ lösen, hält man sich vor Augen, was sich in der rekursiven Variante abspielt: der Aufruf übergibt die aktuellen Parameter an die formalen und die Kontrolle geht an den rekursiven Aufruf über und wenn der abgearbeitet ist, kommt der zweite Aufruf an die Reihe.

Wenn wir uns keine rekursiven Aufruf erlauben sondern das Ablaufgeschehen durch eine Schleife steuern, dann können wir natürlich keine Parameter an eine Prozedur "übergeben" die sie *lokal* abarbeiten kann und also dürfen wir uns nicht an die Abarbeitung der ersten Teilaufgabe machen ohne uns die aktuellen Parameter des zweiten Aufrufs *zu merken*. Dazu verwendet man einen *Stack* (nichts anderes passiert selbstverständlich auch zur Laufzeit von Prozedur- und Funktionsaufrufen, deren Parameter mittels des sogenannten *Laufzeitstacks* übergeben werden). Der Stack muß die Parameter aufnehmen, in diesem Fall also ein Paar von Indizes. Zu Beginn werden die Array-Grenzen auf den Stack abgelegt.[1] Die Schleife und das Sortieren endet, wenn alle Aufrufe abgearbeitet sind, d.h., der Stack leer ist. Das Ergebnis steht in Abbildung 3.

```
let qsort_a_i (a : int array) =
  let (s : (int * int) Stack.t) = Stack.create()
  and partition ((p: int),(r: int)) =
    (if (p < r)
    then (
      let x = a.(p)                                          (* Pivot      *)
      and i = ref p and j = ref r and h = ref 0 in
      while (!i)<=(!j)
      do
        while a.(!i) < x  do i := !i + 1; done;
        while a.(!j) > x  do j := !j - 1; done;
        if (!i)<(!j)
        then
           (h := a.(!i); a.(!i) <- a.(!j); a.(!j) <- !h; (* Austausch *)
            i := !i+1; j:=!j-1)
        else (i:=!i+1)
      done;
      !j)                                                    (* Rueckgabe *)
    else p)
  in
  let p = ref 0 and r = ref ((Array.length a) - 1) and q = ref 0
  in
  Stack.push (!p, !r) s;
  while Stack.length s > 0
  do
    let (p,r) = Stack.pop s
    in
    q := partition (p,r);
    if (p < !q) then  Stack.push (   p,!q) s else ();
    if (!q<  r) then  Stack.push (!q+1, r) s else ();
  done;;
```

Abbildung 3: Iterative Version von Quicksort

Für die Sequenz $(9, 2, 4, 1, 8, 6, 5, 7, 3, 0)$ ergibt sich folgender dynamischer Verlauf des Stacks:

[1] Wir gehen hier von Array-Indizes beginnend bei 0 aus.

```
(0,9)
(9,9),(0,8)
(0,8)
(1,8)
(2,8)
(4,8),(2,3)
(8,8),(4,7),(2,3)
(4,7),(2,3)
(5,7),(2,3)
(6,7),(2,3)
(7,7),(2,3)
(2,3)
(3,3)
```

Die Auflösung von Rekursion über Stacks ist allgemeingültig und unabhängig vom Beispiel Quicksort. Man sieht auch, warum man bei *endrekursiven* Prozeduren den Stack nicht wirklich braucht: da es immer nur *einen* aktuell rekursiven Aufruf gibt, muß man sich keine weiteren Parameter für später merken, anders ausgedrückt, da der Stack immer von der Länge eins (oder kleiner am Ende) ist, braucht man die dynamische Stack-Struktur nicht. Dies wird von Compilern in der Regel automatisch berücksichtigt.