Technische Universität München
Fakultät für Informatik
R. Kuntschke, I. Nagy,
Dr. M. Schneider

SS 2006
Lösungsvorschläge zu Blatt 6
16. Juni 2006

Übungen zu Einführung in die Informatik I

Aufgabe 21 Terminierung (Lösungsvorschlag)

Zur Lösung der vier Teilaufgaben suchen wir Eingabewerte, für die es möglich ist Abstiegsfunktionen, wie in der Vorlesung definiert, anzugeben.

a)
- `f_a(m,n)` terminiert für alle `n` $\in \mathbb{N}$:
  Zum Nachweis der Terminierung für diese Werte wählen wir die Abstiegsfunktion
  $$h : \mathbb{Z} \times \mathbb{N} \to \mathbb{N},\ \ h(m,n) = n.$$
  Es treten nur für `n` $\neq 0$ rekursive Aufrufe auf, d.h. der Parameter `n-1` des rekursiven Aufrufs ist nichtnegativ und es gilt:
  $$h(m+1,n-1) = h(m+2,n-1) = n-1 < n = h(n)$$
- `f_a(m,n)` terminiert nicht, falls `n` $\in \mathbb{Z} \setminus \mathbb{N}$:
  Für `n` $\in \mathbb{Z} \setminus \mathbb{N}$ ist `n`, `n-1`, `n-2`, `n-3`, ... eine unendliche Folge, in der 0 nicht auftritt. Die durch den Aufruf `f_a(m,n)` angestoßene Folge rekursiver Aufrufe `f_a(m+1,n-1)`, `f_a(m+2,n-2)`, `f_a(m+3,n-3)`, ... erreicht also niemals den Terminierungsfall `f_a(*,0)`. Dabei steht `*` für einen belieben Wert des ersten Parameters.

b)
- `f_b(m,n)` terminiert für alle `(m,n)` $\in \mathbb{Z} \times \mathbb{Z}$ mit `m` $\leq$ `n`:
  Zum Nachweis der Terminierung für diese Werte wählen wir die Abstiegsfunktion
  $$h : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N},\ \ h(m,n) = n - m.$$
  Es treten nur für `n` $-$ `m` $\neq 0$, d.h. für `n` $-$ `m` > 0 im Fall `m` $\leq$ `n`, rekursive Aufrufe auf und es gilt:
  $$h(m,n-1) = n-1-m = n-m-1 = h(m,n)-1 < h(m,n)$$
  $$h(m+1,n) = n-(m+1) = n-m-1 = h(m,n)-1 < h(m,n)$$

- `f_b(m,n)` terminiert nicht, falls `m` $>$ `n`:
  Für `m` $>$ `n` ist `n`, `n` -1, `n` -2, `n` $-3, \ldots$ eine unendliche Folge, in der `m` nicht auftritt. Die durch den Aufruf `f_b(m,n)` angestoßene Folge rekursiver Aufrufe `f_b(m,n-1)`, `f_b(m,n-2)`, `f_b(m,n-3)`, ... erreicht also niemals den Terminierungsfall `m == n`.

c)
- `f_c(m,n)` terminiert für alle `(m,n)` $\in \mathbb{Z} \times \mathbb{Z}$ mit `m` $\leq$ `n` und (`n` $-$ `m`) *mod* $2 = 0$:
  Zum Nachweis der Terminierung für diese Werte wählen wir die Abstiegsfunktion
  $$h : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N},\ \ h(m,n) = n - m.$$
  Es treten nur für `n` $-$ `m` $\neq 0$ rekursive Aufrufe auf und es gilt:
  $$h(m+1,n-1) = n-1-(m+1) = n-1-m-1 = h(m,n)-2 < h(m,n)$$
  (**Achtung**: Die Voraussetzung (`n` $-$ `m`) *mod* $2 = 0$ stellt sicher, dass mit $h(m,n)$ auch $h(m+1,n-1)$ in $\mathbb{N}$ liegt.)

- `f_c(m,n)` terminiert nicht, falls $\texttt{m} \leq \texttt{n}$ und $(\texttt{n}-\texttt{m})$ *mod* $2 = 1$:
  Falls gilt $(\texttt{n}-\texttt{m})$ *mod* $2 = 1$ (d.h. falls die Differenz $\texttt{n}-\texttt{m}$ ungerade ist), dann bildet $\texttt{n}-\texttt{m}$, $(\texttt{n}-1)-(\texttt{m}+1)$, $(\texttt{n}-2)-(\texttt{m}+2)$, $(\texttt{n}-3)-(\texttt{m}+3)$, ... eine unendliche Folge ungerader Zahlen, in denen die 0 nicht auftritt. Die durch den Aufruf `f_c(m,n)` angestoßene Folge rekursiver Aufrufe `f_c(m+1,n-1)`, `f_c(m+2,n-2)`, `f_c(m+3,n-3)`,... erreicht also niemals den Terminierungsfall `m==n`.
- `f_c(m,n)` terminiert nicht, falls $\texttt{m} > \texttt{n}$:
  Für $\texttt{m} > \texttt{n}$ ist $\texttt{n}-\texttt{m}$, $(\texttt{n}-1)-(\texttt{m}+1)$, $(\texttt{n}-2)-(\texttt{m}+2)$, $(\texttt{n}-3)-(\texttt{m}+3)$, ... eine unendliche Folge, in der die 0 nicht auftritt. Die durch den Aufruf `f_c(m,n)` angestoßene Folge rekursiver Aufrufe `f_c(m+1,n-1)`, `f_c(m+2,n-2)`, `f_c(m+3,n-3)`,... erreicht also niemals den Terminierungsfall `m==n`.

d) `f_d(n)` terminiert für alle $\texttt{n} \in \mathbb{Z}$:

Zum Nachweis der Terminierung für diese Werte wählen wir die Abstiegsfunktion

$$h : \mathbb{Z} \to \mathbb{N},\ \ h(n) = |100 - n|.$$

Es treten nur für $\texttt{n} \neq 100$ rekursive Aufrufe auf und es gilt:

$$\begin{aligned}
\texttt{n} < 100 \Rightarrow h(200-n-1) &= |100-(200-n-1)| = (200-n-1)-100 = 99-n \\
&< 100-n = |100-n| = h(n) \\
\texttt{n} > 100 \Rightarrow h(200-n+1) &= |100-(200-n+1)| = 100-(200-n+1) = -101+n \\
&< -100+n = |100-n| = h(n)
\end{aligned}$$

Aufgabe 22 Terminierung (Lösungsvorschlag)

Zur Lösung der Aufgabe suchen wir Eingabewerte, für die es möglich ist Abstiegsfunktion, wie in der Vorlesung definiert, anzugeben:

`f(n,m,k)` terminiert für alle $(\texttt{m},\texttt{n},\texttt{k}) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$:

Zum Nachweis der Terminierung für diese Werte wählen wir die Abstiegsfunktion

$$h : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N},\ \ h(m,n) = |m-n|.$$

Es treten nur für $\texttt{n} \neq m$ rekursive Aufrufe auf und es gilt:

$$\begin{aligned}
\texttt{n<m} \Rightarrow h(2\cdot m-n-1,m) &= |m-(2\cdot m-n-1)| = |-m+n+1| \\
&= (m-n-1) < (m-n) = |m-n| = h(n,m) \\
\texttt{n>m} \Rightarrow h(2\cdot m-n+1,m) &= |m-(2\cdot m-n+1)| = |-m+n-1| \\
&= (-m+n-1) < (-m+n) = |m-n| = h(n,m)
\end{aligned}$$

Aufgabe 23 Ordnungen über Zeichensequenzen (Lösungsvorschlag)

a) Es wird gezeigt, dass die Präfixrelation reflexiv, antisymmetrisch und transitiv ist:

Reflexivität: Sei $x \in C^*$. Da $x = x \circ \varepsilon$ gilt für alle $x \in C^*$ folgt $x \sqsubseteq x$ und die Präfixrelation ist reflexiv.

Antisymmetrie: Sei $x, y \in C^*$; außerdem gelte $x \sqsubseteq y$ und $y \sqsubseteq x$. Dann gibt es ein $u, u' \in C^*$ derart, dass $y = x \circ u$ und $x = y \circ u'$. Daraus folgt $y = y \circ u \circ u'$ und $u = u' = \varepsilon$. Damit gilt $y = x \circ \varepsilon$ und schließlich $y = x$. Die Präfixrelation ist somit antisymmetrisch.

Transitivität: Sei $x, y, z \in C^*$; außerdem gelte $x \sqsubseteq y$ und $y \sqsubseteq z$. Dann gibt es ein $v, v' \in C^*$ derart, dass $y = x \circ v$ und $z = y \circ v'$. Wir erhalten also $z = x \circ v \circ v'$. Mit $v'' = v \circ v'$ folgt $z = x \circ v''$ oder $x \sqsubseteq z$. Die Präfixrelation ist somit transitiv.

Insgesamt ist also gezeigt, dass die Präfixrelation eine Halbordnung ist.

b) Totalität, Wohlordnung und Fundiertheit der Präfixrelation:

Wir zeigen durch Angabe eines Gegenbeispiels: Die Präfixrelation ist keine totale Ordnung. Sei $C = \{a, b\}$ und $x, y \in C^*$ mit $x = a$ und $y = b$. Dann gilt weder $x \sqsubseteq y$ noch $y \sqsubseteq x$ und die Präfixrelation ist nicht total.

Eine partielle Ordnung heißt bekanntlich fundiert, wenn jede (echt) absteigende Kette endlich ist. Wenn y ein echtes Präfix von x ist, dann ist u in der Zerlegung $x = y \circ u$ nicht leer, die Länge von y ist also echt kleiner als die von x. Jede echt absteigende Kette von Präfixen von x enthält also höchstens $|x| + 1$ Elemente.

Eine partielle Ordnung heißt wohlgeordnet, wenn jede Teilmenge ein kleinstes Element hat. Wichtig ist hierbei, dass ein kleinstes Element mit allen anderen Elementen vergleichbar, d.h. in Relation steht. Nun gibt es aber in der Praefixrelation Teilmengen $N \subset C^*$ die zwar minimale, nicht jedoch ein kleinstes Element enthalten; Beispiel $N = \{aabb, aa, b, bb\}$; hierbei sind aa und b minimal, d.h. minimale Elemente einer absteigenden Kette, jedoch unvergleichbar hinsichtlich der Praefixrelation. Die Praefixrelation ist also nicht wohlgeordnet wohl aber fundiert.

c) Die lexikographische Ordnung ist nicht fundiert, denn für $a \leq_C b$ gibt es unendlich viele Wörter ax, $x \in C^*$ zwischen a und b und somit unendlich absteigende Ketten.

d) Eine fundierte totale Ordnung über C^* lässt sich folgendermaßen definieren: x ist kleiner gleich y, falls $|x| \leq |y|$ oder falls $|x| = |y|$ und x ist lexikographisch kleiner gleich y.

Aufgabe 24 Vergleich von Bäumen (Lösungsvorschlag)

Zur Lösung wird die bereits bekannte Datenstruktur

```
type 'a btree = Empty | Node of ('a * 'a btree * 'a btree);;
```

und die Funktion `in_order` aus Aufgabe 23 verwendet.

a) Aus der einfachen Darstellung des Baumes in Listenform (z. B. mit `in_order`) reicht nicht aus um die identische Struktur des Baumes zu gewährleisten. Erst der Vergleich von verschiedenen Reihenfolgen (z. B. `in_order` und `pre_order`) stellt dies sicher.

```
# let sample_tree =
  Node((13,7),
      Node((12,7), Node((40,34),Empty,Node((50,65),Empty,Empty)),
              Node((67,92),Node((50,65),Empty,Empty),Empty)),
      Node((22,3), Node((22,54),Empty,Empty),
               Node((26,53),Empty,Node((50,29),Empty,Empty))));;
# let sample_tree2 =
  Node((13, 7),
      Node((12,7), Node((40,34),Node((50,65),Empty,Empty),Empty),
              Node((67,92),Node((50,65),Empty,Empty),Empty)),
```

```
        Node((22,3), Node((22,54),Empty,Empty),
                 Node ((26,53),Empty,Node((50,29),Empty,Empty))));;

# let compare_tree tree1 tree2 =
        let tree1_list_io = in_order tree1 in
        let tree2_list_io = in_order tree2 in
        let tree1_list_po = pre_order tree1 in
        let tree2_list_po = pre_order tree2 in
        let rec compare_list list1 list2 =
                match (list1,list2) with
                  ([],[]) -> true
                | (x::xl, y::yl) when (x=y) -> compare_list xl yl
                | _ -> false
        in (compare_list tree1_list_io tree2_list_io)
         && (compare_list tree1_list_po tree2_list_po);;
val compare_tree : 'a btree -> 'a btree -> bool = <fun>
# compare_tree sample_tree sample_tree2;;
- : bool = false
# compare_tree sample_tree sample_tree;;
- : bool = true
```

b)

```
# let rec compare_tree2 tree1 tree2 =
        match (tree1,tree2) with
          (Empty,Empty) -> true
        | (Node(x,left1,right1),Node(y,left2,right2)) when (x=y) ->
                (compare_tree2 left1 left2) && (compare_tree2 right1 right2)
        | _ -> false;;
val compare_tree2 : 'a btree -> 'a btree -> bool = <fun>
# compare_tree2 sample_tree sample_tree2;;
- : bool = false
# compare_tree2 sample_tree sample_tree;;
- : bool = true
```

Aufgabe 25 Tiefensuche in Graphen und Straßennetze (Lösungsvorschlag)

Eine mögliche Lösung könnte folgendermaßen aussehen:

```
type stadt = Muenchen|Augsburg|Koeln|Stuttgart|Nuernberg|Frankfurt|Berlin|Hamburg;;

type knoten = Knoten of (stadt * bool);;

let staedte = [Knoten(Muenchen, false); Knoten(Augsburg, false);
       Knoten(Stuttgart, false); Knoten(Nuernberg, false);
       Knoten(Frankfurt, false); Knoten(Berlin, false);
       Knoten(Hamburg, false); Knoten(Koeln, false)];

type kante = Kante of (stadt * stadt);;

let wege = [Kante(Muenchen, Augsburg); Kante(Augsburg, Stuttgart);
    Kante(Nuernberg, Muenchen); Kante(Berlin, Hamburg);
    Kante(Koeln, Muenchen); Kante(Nuernberg, Frankfurt)];;
```

```
let push(elem, stack) = elem::stack;;

let pop(s) = match s with
| head::tail -> tail
| [] -> [];;

let top(s) = match s with
| head::tail -> [head]
| [] -> [];;

let expandiere_knoten(graph, quelle) =
  let rec expandiere_knoten_em(g, q, n) = match g with
  | h::t -> ( match h with
       | Kante(von, nach) when (von = q) -> expandiere_knoten_em(t, q, n@[nach])
       | Kante(von, nach) when (nach = q) -> expandiere_knoten_em(t, q, n@[von])
       | Kante(_, _) -> expandiere_knoten_em(t, q, n))
  | [] -> n
  in expandiere_knoten_em(graph, quelle, []);;

let markiere_knoten(knotenliste, knoten) =
  let rec markiere_knoten_em(knotenliste, knoten, markiert) =
    match knotenliste with
    | h::t -> ( match h with
      | Knoten(k, m) when (k = knoten)
        -> markiere_knoten_em(t, knoten, markiert@[Knoten(k, true)])
      | Knoten(k, m)
        -> markiere_knoten_em(t, knoten, markiert@[Knoten(k, m)]))
    | [] -> markiert
  in markiere_knoten_em(knotenliste, knoten, []);;

let rec knoten_markiert(knotenliste, knoten) =
  match knotenliste with
  | h::t -> ( match h with
    | Knoten(k, m) ->
        if (m = true) && (k = knoten) then true
        else knoten_markiert(t, knoten))
  | [] -> false;;

let dfs(knoten, kanten, quelle, ziel) =
  let rec iteriere_stack(knotenstack, markierte_knoten) =
    match top(knotenstack) with
    | [e] -> if (e = ziel) then true
    else
        let rec iteriere_nachbarn(nachbarn, st, mk) = match nachbarn with
        | h :: t when knoten_markiert(mk, h) ->
            iteriere_nachbarn(t, st, mk)
        | h :: t when not(knoten_markiert(mk, h)) ->
            iteriere_nachbarn(t, push(h, st), mk)
        | [] -> iteriere_stack(st, mk)
        in iteriere_nachbarn(expandiere_knoten(kanten, e),
                             pop(knotenstack),
                             markiere_knoten(markierte_knoten, e))
    | [] -> false
```

```
  in iteriere_stack(push(quelle, []), knoten);;

dfs(staedte, wege, Stuttgart, Frankfurt);;
dfs(staedte, wege, Stuttgart, Berlin);;
```