Technische Universität München
Fakultät für Informatik
D. Gmach, M. Rickert,
Dr. M. Schneider

WS 2005/2006
Lösungsvorschläge zu Blatt 6
02. Dezember 2005

Übungen zu Einführung in die Informatik I

Aufgabe 23 Bäume in OCaml (Lösungsvorschlag)

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

# let comp_xy (a,b) (c,d) = if (a < c) then -1
  else if (a = c && b < d) then -1
  else if (a = c && b = d) then 0
  else 1;;
val comp_xy : 'a * 'b -> 'a * 'b -> int = <fun>
# comp_xy (1,4) (2,3);;
- : int = -1
# comp_xy (2,3) (2,3);;
- : int = 0
# comp_xy (2,4) (2,3);;
- : int = 1
```

b)
```
# let rec in_order btree = match btree with
          Empty -> []
        | Node(e, left, right) ->
              (in_order left) @ [e] @ (in_order right);;
val in_order : 'a btree -> 'a list = <fun>

(* Beispieleingabe zum Testen)
# 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))));;
val sample_tree : (int * int) btree =
  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))))
```

```
(* Testen *)
# in_order sample_tree;;
- : (int * int) list =
[(40, 34); (50, 65); (12, 7); (50, 65); (67, 92); (13, 7); (22, 54);
 (22, 3); (26, 53); (50, 29)]
```

c) Die Implementierung dieser Funktion wird durch die Verwendung der Funktion `in_order` deutlich vereinfacht.

```
(* Sortiertes einfügen *)
# let rec insert f elem tree =
        match tree with
            Empty -> Node(elem, Empty, Empty)
          | Node(x, left, right) when (f x elem = 0) ->
                Node(x, left, right)
          | Node(x, left, right) when (f x elem > 0) ->
                Node(x, (insert f elem left), right)
          | Node(x, left, right) ->
                Node(x, left, (insert f elem right));;

# let rec sort_tree f btree =
        let list = in_order btree in
        let rec insert_rec l = match l with
                [] -> Empty
                | x::xs -> insert f x (insert_rec xs) in
        insert_rec list;;
val sort_tree : ('a -> 'a -> int) -> 'a btree -> 'a btree = <fun>

# sort_tree comp_xy sample_tree;;
- : (int * int) btree =
Node((50, 29),
     Node((26, 53),
          Node((22, 54),
               Node((22, 3),
                    Node ((12, 7),
                          Empty,
                          Node ((13, 7), Empty, Empty)),
                    Empty),
               Empty),
          Node ((40, 34), Empty, Empty)),
     Node ((50, 65),
           Empty,
           Node ((67, 92), Empty, Empty)))

# in_order (sort_tree comp_xy sample_tree);;
- : (int * int) list =
[(12, 7); (13, 7); (22, 3); (22, 54); (26, 53); (40, 34); (50, 29); (50, 65);
 (67, 92)]
```

d)
```
# let rec pre_order btree = match btree with
          Empty -> []
```

```
        | Node(e, left, right) -> e :: (pre_order left) @ (pre_order right);;
val in_order : 'a btree -> 'a list = <fun>

# pre_order sample_tree;;
- : (int * int) list =
[(13, 7); (12, 7); (40, 34); (50, 65); (67, 92); (50, 65); (22, 3); (22, 54);
 (26, 53); (50, 29)]

# let rec post_order btree = match btree with
          Empty -> []
        | Node(e, left, right) -> (post_order left) @ (post_order right) @ [e];;
val post_order : 'a btree -> 'a list = <fun>
# post_order sample_tree;;
- : (int * int) list =
[(50, 65); (40, 34); (50, 65); (67, 92); (12, 7); (22, 54); (50, 29);
 (26, 53); (22, 3); (13, 7)]
```

e)
```
# let rec map_tree f btree = match btree with
          Empty -> Empty
        | Node(e, left, right) -> Node(f e, map_tree f left, map_tree f right);;
val map_tree : ('a -> 'b) -> 'a btree -> 'b btree = <fun>
# map_tree (fun (x,y) -> x * 1000 + y) sample_tree;;
- : int btree =
Node
 (13007,
  Node
   (12007, Node (40034, Empty, Node (50065, Empty, Empty)),
    Node (67092, Node (50065, Empty, Empty), Empty)),
  Node
   (22003, Node (22054, Empty, Empty),
    Node (26053, Empty, Node (50029, Empty, Empty))))
```

Aufgabe 24 Verifikation funktionaler Programme

a)
- $\texttt{xs}_0 = []$: Gemäß der angegebenen Definition gilt dann $\texttt{rev}\ \texttt{xs}_0 = []$. Somit folgt:

$$\begin{aligned} \texttt{rev}\,(\texttt{app}\ \texttt{xs}_0\ \texttt{ys}) &= \texttt{rev}\,(\texttt{app}\ []\ \texttt{ys}) \\ &= \texttt{rev}\ \texttt{ys} \\ &= \texttt{app}\,(\texttt{rev}\ \texttt{ys})\ [] \\ &= \texttt{app}\,(\texttt{rev}\ \texttt{ys})\,(\texttt{rev}\ \texttt{xs}_0) \quad \square \end{aligned}$$

  Bemerkung: Der Schritt von Zeile 2 nach Zeile 3 wurde in der Vorlesung bewiesen.

- **Induktionsschluss:** Wir betrachten die Liste $\texttt{xs}' = \texttt{x} :: \texttt{xs}$. Nach Induktionsannahme ist das zu beweisende Prädikat für Listen mit einer Länge $l < \mathit{length}(\mathit{xs}')$ gültig, d.h. die Aussage $\texttt{rev}\,(\texttt{app}\ \texttt{xs}\ \texttt{ys}) = \texttt{app}\,(\texttt{rev}\ \texttt{ys})\,(\texttt{rev}\ \texttt{xs})$ ist gültig. Wir bilden nun $\texttt{rev}(\texttt{app}\ \texttt{xs}'\ \texttt{ys})$. Dann gilt (unter Verwendung der bisher gezeigten Beziehungen):

$$\begin{aligned} \texttt{rev}\,(\texttt{app}\ \texttt{xs}'\ \texttt{ys}) &\overset{Def.}{=} \texttt{rev}\,(\texttt{x} :: (\texttt{app}\ \texttt{xs}\ \texttt{ys})) \\ &\overset{Def.}{=} \texttt{app}\,(\texttt{rev}\,(\texttt{app}\ \texttt{xs}\ \texttt{ys}))\,[\texttt{x}] \end{aligned}$$

$$\begin{aligned}
&\overset{IA}{=} && \texttt{app}\ (\texttt{app}\ (\texttt{rev}\ \texttt{ys})\ (\texttt{rev}\ \texttt{xs}))\ [\texttt{x}] \\
&\overset{Ass.}{=} && \texttt{app}\ (\texttt{rev}\ \texttt{ys})\ (\texttt{app}\ (\texttt{rev}\ \texttt{xs})\ [\texttt{x}]) \\
&\overset{s.o.}{=} && \texttt{app}\ (\texttt{rev}\ \texttt{ys})\ (\texttt{rev}\ (\texttt{x} :: \texttt{xs})) \\
&= && \texttt{app}\ (\texttt{rev}\ \texttt{ys})\ (\texttt{rev}\ \texttt{xs}')\ \ \square
\end{aligned}$$

Bemerkung: Die Assoziativität von `app` wurde in der Vorlesung bewiesen.

b) Zu beweisen ist nun das Prädikat

$$\texttt{rev}\ (\texttt{rev}\ \texttt{xs}) = \texttt{xs}$$

Der Beweis erfolgt mittels Induktion:

Induktionsanfang: $\texttt{xs} = []$. Dann gilt gemäß Definition der Funktion `rev`:

$$\begin{aligned}
\texttt{rev}\ (\texttt{rev}\ \texttt{xs}) &= \texttt{rev}\ (\texttt{rev}\ []) \\
&= \texttt{rev}\ [] \\
&= [] \ \square
\end{aligned}$$

Induktionsschluss: Es wird gezeigt, dass aus der Gültigkeit von $\texttt{rev}\ (\texttt{rev}\ \texttt{xs}) = \texttt{xs}$ die Gültigkeit von $\texttt{rev}\ (\texttt{rev}\ (\texttt{x} :: \texttt{xs})) = \texttt{x} :: \texttt{xs}$ folgt:

$$\begin{aligned}
\texttt{rev}\ (\texttt{rev}\ (\texttt{x} :: \texttt{xs})) &= \texttt{rev}\ (\texttt{app}\ (\texttt{rev}\ \texttt{xs})\ [\texttt{x}]) \\
&\overset{a)}{=} \texttt{app}\ (\texttt{rev}\ [\texttt{x}])\ (\texttt{rev}\ (\texttt{rev}\ \texttt{xs})) \\
&\overset{IA}{=} \texttt{app}\ (\texttt{rev}\ [\texttt{x}])\ \texttt{xs} \\
&= \texttt{x} :: \texttt{xs}\ \ \square
\end{aligned}$$

Aufgabe 25 Verifikation funktionaler Programme (Lösungsvorschlag)

Der Beweis der Behauptung

```
length (app ks ls) = length ls + length ks        (1)
```

erfolgt durch Induktion über die Länge von `ls`.

Induktionsanfang: Sei `ls = []`. Dann folgt

$$\begin{aligned}
\texttt{length (app [] ks)} &\overset{Def.}{=} \texttt{length ks} \\
&= 0 + \texttt{length ks} \\
&= \texttt{length []} + \texttt{length ks} \\
&= \texttt{length ls} + \texttt{length ks}
\end{aligned}$$

Induktionsschluss: Sei `ls' = l :: ls`. Dann folgt:

$$\begin{aligned}
\texttt{length (app (l :: ls) ks)} &\overset{Def.}{=} \texttt{length l :: (app ls ks)} \\
&\overset{Def.}{=} 1 + \texttt{length (app ls ks)} \\
&\overset{IV}{=} 1 + \texttt{length ls} + \texttt{length ks} \\
&\overset{Def.}{=} \texttt{length (l :: ls)} + \texttt{length ks}
\end{aligned}$$

Aufgabe 26 Existenz- und Allquantor (Lösungsvorschlag)

Eine Funktion, die den „Existenzquantor“ für Sequenzen modelliert lautet:

```
# let rec existiert f l  =
  match f, l with
    (g, [])               -> false
  | (g, h::tail) when g h -> true
  | (g, h::tail)          -> existiert g tail ;;
val existiert : ('a -> bool) -> 'a list -> bool = <fun>
```

Für den „Allquantor“ ergibt sich analog:

```
# let rec fuer_alle f l =
  match f, l with
    (g, [])                    -> true
  | (g, h::tail) when not(g h) -> false
  | (g, h::tail)               -> fuer_alle g tail ;;
val fuer_alle : ('a -> bool) -> 'a list -> bool = <fun>
```