Allgemeiner:

`app (rev x) y = rev1 x y` für alle Listen `x, y`.

Beweis: Induktion nach der Länge n von `x`

$n = 0$: Dann gilt: `x = []`. Wir schließen:

```
app (rev x) y  =  app (rev []) y
               =  app (match [] with [] -> [] | ...) y
               =  app [] y
               =  y
               =  match [] with [] -> y | ...
               =  rev1 [] y
               =  rev1 x y      :-)
```

$n > 0$: Dann gilt: `x = h::t` wobei `t` Länge $n - 1$ hat.

Wir schließen (unter Weglassung einfacher Zwischenschritte):

```
app (rev x) y  =  app (rev (h::t)) y
               =  app (app (rev t) [h]) y
               =  app (rev t) (app [h] y)   wegen Beispiel 1
               =  app (rev t) (h::y)
               =  rev1 t (h::y)             nach Induktionsvoraussetzung
               =  rev1 (h::t) y
               =  rev1 x y                  :-))
```

Diskussion:

- Wieder haben wir implizit die Terminierung der Funktionsaufrufe von `app`, `rev` und `rev1` angenommen :-)
- Deren Terminierung können wir jedoch leicht mittels Induktion nach der Tiefe des ersten Arguments nachweisen.
- Die Behauptung:

  ```
  rev x = rev1 x []
  ```

  folgt aus:

  ```
  app (rev x) y = rev1 x y
  ```

  indem wir: `y = []` setzen und Aussage (1) aus Beispiel 1 benutzen :-)

Beispiel 3:

```
let rec sorted = fun x -> match x
        with x1::x2::xs -> (match x1 <= x2
                        with true -> sorted (x2::xs)
                          | false -> false)
           | _           -> true

    and merge = fun x -> fun y -> match (x,y)
        with ([],y) -> y
        |    (x,[]) -> x
        | (x1::xs,y1::ys) -> (match x1 <= y1
                        with true -> x1 :: merge xs y
                          | false -> y1 :: merge x ys
```

Behauptung:

$$\texttt{sorted x} \wedge \texttt{sorted y} \rightarrow \texttt{sorted (merge x y)}$$

für alle Listen `x, y`.

Beweis: Induktion über die Summe n der Längen von `x, y` :-)

Gelte `sorted x` $\wedge$ `sorted y`.

$n = 0$: Dann gilt: `x = [] = y`

Wir schließen:

$$\begin{aligned} \texttt{sorted (merge x y)} &= \texttt{sorted (merge [] [])} \\ &= \texttt{sorted []} \\ &= \texttt{true} \quad \text{:-)} \end{aligned}$$

$n > 0$:

Fall 1: `x = [].`

Wir schließen:

```
sorted (merge x y)   =   sorted (merge [] y)
                     =   sorted y
                     =   true     :-)
```

Fall 2: `y = []` analog :-)

Fall 3: `x = x1::xs` $\wedge$ `y = y1::ys` $\wedge$ `x1` $\leq$ `y1`.

Wir schließen:

```
sorted (merge x y)   =   sorted (merge (x1::xs) (y1::ys))
                     =   sorted (x1 :: merge xs y)
                     =   ...
```

Fall 3.1: `xs = []`

Wir schließen:

```
                ...  =   sorted (x1 :: merge [] y)
                     =   sorted (x1 :: y)
                     =   sorted y
                     =   true     :-)
```

Fall 3.2: `xs = x2::xs'` $\wedge$ `x2` $\leq$ `y1`.

Insbesondere gilt: `x1` $\leq$ `x2` $\wedge$ `sorted xs`.

Wir schließen:

```
...  =  sorted (x1 :: merge (x2::xs') y)
     =  sorted (x1 :: x2 :: merge xs' y)
     =  sorted (x2 :: merge xs' y)
     =  sorted (merge xs y)
     =  true   nach Induktionsannahme   :-)
```

Fall 3.3: `xs = x2::xs'` $\wedge$ `x2` $>$ `y1`.

Insbesondere gilt: `x1` $\leq$ `y1` $<$ `x2` $\wedge$ `sorted xs`.

Wir schließen:

```
...  =  sorted (x1 :: merge (x2::xs') (y1::ys))
     =  sorted (x1 :: y1 :: merge xs ys)
     =  sorted (y1 :: merge xs ys)
     =  sorted (merge xs y)
     =  true   nach Induktionsannahme   :-)
```

Fall 4: `x = x1::xs` $\wedge$ `y = y1::ys` $\wedge$ `x1` $>$ `y1`.

Wir schließen:

```
sorted (merge x y)  =  sorted (merge (x1::xs) (y1::ys))
                    =  sorted (y1 :: merge x ys)
                    =  ...
```

Fall 4.1: `ys = []`

Wir schließen:

```
                ...  =  sorted (y1 :: merge x [])
                     =  sorted (y1 :: x)
                     =  sorted x
                     =  true     :-)
```

Fall 4.2: `ys = y2::ys'` $\wedge$ `x1` $>$ `y2`.

Insbesondere gilt: `y1` $\leq$ `y2` $\wedge$ `sorted ys`.

Wir schließen:

```
...  =  sorted (y1 :: merge x (y2::ys'))
     =  sorted (y1 :: y2 :: merge x ys')
     =  sorted (y2 :: merge x ys')
     =  sorted (merge x ys)
     =  true   nach Induktionsannahme   :-)
```

Fall 4.3: `ys = y2::ys'` $\wedge$ `x1` $\leq$ `y2`.

Insbesondere gilt: `y1` $<$ `x1` $\leq$ `y2` $\wedge$ `sorted ys`.

Wir schließen:

```
...  =  sorted (y1 :: merge (x1::xs) (y2::ys'))
     =  sorted (y1 :: x1 :: merge xs ys)
     =  sorted (x1 :: merge xs ys)
     =  sorted (merge x ys)
     =  true   nach Induktionsannahme   :-))
```

Diskussion:

- Wieder steht der Beweis unter dem Vorbehalt, dass alle Aufrufe der Funktionen `sorted` und `merge` terminieren :-)
- Als zusätzliche Technik benötigten wir Fallunterscheidungen über die verschiedenen Möglichkeiten für Argumente in den Aufrufen :-)
- Die Fallunterscheidungen machten den Beweis länglich :-(

  // Der Fall $n = 0$ ist tatsächlich überflüssig,

  // da er in den Fällen 1 und 2 enthalten ist :-)

7 Parallele Programmierung

Die Bibliothek threads.cma unterstützt die Implementierung von Systemen, die mehr als einen Thread benötigen :-)

Beispiel:

```
module Echo = struct open Thread
   let echo () = print_string (read_line () ^ "\n")
   let main    =    let t1 = create echo ()
                  in join t1;
                     print_int (id (self ()));
                     print_string "\n"
end
```

Kommentar:

- Die Struktur `Thread` versammelt Grundfunktionalität zur Erzeugung von Nebenläufigkeit :-)
- Die Funktion `create: ('a -> 'b) -> 'a -> t` erzeugt einen neuen Thread mit den folgenden Eigenschaften:
  - der Thread wertet die Funktion auf dem Argument aus;
  - der erzeugende Thread erhält die Thread-Id zurück und läuft unabhängig weiter.
  - Mit den Funktionen: `self : unit -> t` bzw. `id : t -> int` kann man die eigene Thread-Id abfragen bzw. in ein `int` umwandeln.

Weitere nützliche Funktionen:

- Die Funktion `join: t -> unit` hält den aktuellen Thread an, bis die Berechnung des gegebenen Threads beendet ist.
- Die Funktion: `kill: t -> unit` beendet einen Thread;
- Die Funktion: `delay: float -> unit` verzögert den aktuellen Thread um eine Zeit in Sekunden;
- Die Funktion: `exit: unit -> unit` beendet den aktuellen Thread.

Achtung:

- Die interaktive Umgebung funktioniert nicht mit Threads !!
- Stattdessen muss man mit der Option: `-thread` compilieren:

  ```
  > ocamlc -thread unix.cma threads.cma Echo.ml
  ```

- Die Bibliothek `threads.cma` benötigt dabei Hilfsfunktionalität der Bibliothek `unix.cma` :-)

  // unter Windows sieht die Sache vermutlich anders aus :-))
- Das Programm testen können wir dann durch Aufruf von:

  ```
  > ./a.out          :-)
  ```

```
> ./a.out
> abcdefghijk
> abcdefghijk
> 0
>
```

- Ocaml-Threads werden vom System nur simuliert :-(
- Die Erzeugung von Threads ist billig :-))
- Die Programm-Ausführung endet mit der Terminierung des Threads mit der Id `0` .

7.1 Kanäle

Threads kommunizieren über Kanäle :-)

Für Erzeugung, Senden auf und Empfangen aus einem Kanal stellt die Struktur `Event` die folgende Grundfunktionalität bereit:

```
type 'a channel
new_channel : unit -> 'a channel
type 'a event
always  : 'a -> 'a event
sync    : 'a event -> 'a
send    : 'a channel -> 'a -> unit event
receive : 'a channel -> 'a event
```

- Jeder Aufruf `new_channel()` erzeugt einen anderen Kanal.
- Über einen Kanal können beliebige Daten geschickt werden !!!
- `always` wandelt einen Wert in ein Ereignis um.
- Senden und Empfangen sind erzeugen Ereignisse ...
- Synchronisierung auf Ereignisse liefert deren Wert :-)

```
module Exchange = struct open Thread open Event
let thread ch = let x = sync (receive ch)
                in print_string (x ^ "\n");
                   sync (send ch "got it!")
let main = let ch = new_channel () in create thread ch;
           print_string "main is sending ... ";
           sync (send ch "Greetings!");
           print_string ("He " ^ sync (receive ch) ^ "\n")
end
```

Diskussion:

- `sync (send ch str)` macht das Ereignis des Sendens der Welt offenbar und blockiert den Sender, bis jemand den Wert aus dem Kanal ausgelesen hat ...
- `sync (receive ch)` blockiert den Empfänger, bis ein Wert im Kanal enthalten ist. Dann liefert der Ausdruck diesen Wert :-)
- Synchrone Kommunikation ist eine Alternative zum Austausch von Daten zwischen Threads bzw. zur Organisation von Nebenläufigkeit $\Longrightarrow$ Rendezvous
- Insbesondere kann sie benutzt werden, um asynchrone Thread-Kooperation zu implementieren :-)