Durch mehrfache Anwendung der Regel für Funktionsaufrufe können wir zusätzlich eine Regel für Funktionen mit mehreren Argumenten ableiten:

$$\frac{e_0 \;=\; \texttt{fun}\; x_1 \ldots x_k \texttt{->}\, e \qquad e_1, \ldots, e_k \text{ terminieren}}{e_0\; e_1 \ldots e_k \;=\; e\left[e_1/x_1, \ldots, e_k/x_k\right]}$$

Diese abgeleitete Regel macht Beweise etwas weniger umständlich :-)

Regel für Pattern Matching:

$$\frac{e_0 \;=\; []}{\texttt{match}\; e_0 \;\texttt{with}\; [] \texttt{->} e_1 \;\texttt{|}\; \ldots \;\texttt{|}\; p_m \texttt{->} e_m \;=\; e_1}$$

$$\frac{e_0 \text{ terminiert} \qquad e_0 \;=\; e_1' :: e_2'}{\texttt{match}\; e_0 \;\texttt{with}\; [] \texttt{->} e_1 \;\texttt{|}\; x :: xs \texttt{->} e_2 \;=\; e_2[e_1'/x, e_2'/xs]}$$

Regel für Pattern Matching:

$$\frac{e_0 \;=\; []}{\texttt{match}\; e_0 \;\texttt{with}\; [] \texttt{->} e_1 \;\texttt{|}\; \ldots \;\texttt{|}\; p_m \texttt{->} e_m \;=\; e_1}$$

$$\frac{e_0 \text{ terminiert} \qquad e_0 \;=\; e_1' :: e_2'}{\texttt{match}\; e_0 \;\texttt{with}\; [] \texttt{->} e_1 \;\texttt{|}\; x :: xs \texttt{->} e_2 \;=\; e_2[e_1'/x, e_2'/xs]}$$

Diese Regeln wollen wir jetzt anwenden ...

7.3 Beweise für MiniOcaml-Programme

Beispiel 1:

```
let rec app = fun x -> fun y -> match x
                    with [] -> y
                    | x::xs -> x :: app xs y
```

Wir wollen nachweisen:

(1) `app x [] = x` für alle Listen `x`.

(2) `app x (app y z) = app (app x y) z`
für alle Listen `x, y, z`.

Idee: Induktion nach der Länge n von `x`

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

Wir schließen:

```
app x []   =   app [] []
           =   match [] with [] -> [] | h::t -> h :: app t []
           =   []
           =   x      :-)
```

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

Wir schließen:

```
app x []   =   app (h::t) []
           =   match h::t with [] -> [] | h::t -> h :: app t []
           =   h :: app t []
           =   h :: t      nach Induktionsannahme
           =   x           :-))
```

Analog gehen wir für die Aussage (2) vor ...

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

Wir schließen:

```
app x (app y z)  =  app [] (app y z)
                 =  match [] with [] -> app y z | h::t -> ...
                 =  app y z
                 =  app (match [] with [] -> y | ...) z
                 =  app (app [] y) z
                 =  app (app x y) z    :-)
```

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

Wir schließen:

```
app x (app y z)  =  app (h::t) (app y z)
                 =  match h::t with [] -> [] | h::t -> h ::
                                 app t (app y z)
                 =  h :: app t (app y z)
                 =  h :: app (app t y) z   nach Induktionsannahme
                 =  app (h :: app t y) z
                 =  app (match h::t with [] -> []
                                       | h::t -> h :: app t y) z
                 =  app (app (h::t) y) z
                 =  app (app x y) z    :-))
```

Diskussion:

- Zur Korrektheit unserer Induktionsbeweise benötigen wir, dass die vorkommenden Funktionsaufrufe terminieren.
- Im Beispiel reicht es zu zeigen, dass für alle `x, y` ein v existiert mit:

$$\texttt{app x y} \Rightarrow v$$

  ... das haben wir aber bereits bewiesen, natürlich ebenfalls mit Induktion ;-)

Beispiel 2:

```
let rec rev = fun x -> match x
        with [] -> []
        | x::xs -> app (rev xs) [x]
let rec rev1 = fun x -> fun y -> match x
        with [] -> y
        | x::xs -> rev1 xs (x::y)
```

Behauptung:

`rev x = rev1 x []` für alle Listen `x`.

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:

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

für alle Listen `x, y`.

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

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

$\boxed{n = 0:}$ Dann gilt: `x = [] = y`

Wir schließen:

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

$n > 0$:

Fall 1: `x = [].`

Wir schließen:

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

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