Script generated by TTT

Title: Seidl: Info2 (20.01.2017)

Date: Fri Jan 20 08:33:14 CET 2017

Duration: 84:16 min

Pages: 48

Der eingebaute Gleichheits-Operator

$$v = v \;\Rightarrow\; \texttt{true}$$

$$v_1 = v_2 \;\Rightarrow\; \texttt{false}$$

sofern v, v_1, v_2 Werte sind, in denen keine Funktionen vorkommen, und v_1, v_2 syntaktisch verschieden sind.

Beispiel 1

$$\dfrac{\dfrac{\texttt{17+4} \Rightarrow \texttt{21} \qquad \texttt{21} \Rightarrow \texttt{21}}{\texttt{17 + 4 = 21} \Rightarrow \texttt{true}} \qquad \texttt{21=21} \Rightarrow \texttt{true}}{}$$

Beispiel 2

```
let f = fun x -> x+1
let s = fun y -> y*y
```

$$\dfrac{\dfrac{\texttt{f = fun x -> x+1}}{\texttt{f} \Rightarrow \texttt{fun x -> x+1}} \qquad 16+1 \Rightarrow 17}{\texttt{f 16} \Rightarrow 17} \qquad \dfrac{\dfrac{\texttt{s = fun y -> y*y}}{\texttt{s} \Rightarrow \texttt{fun y -> y*y}} \qquad 2*2 \Rightarrow 4}{\texttt{s 2} \Rightarrow 4} \qquad 17+4 \Rightarrow 21$$

$$\texttt{f 16 + s 2} \Rightarrow 21$$

// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen

Beispiel 2

```
let f = fun x -> x+1
let s = fun y -> y*y
```

$$\dfrac{\dfrac{\texttt{f = fun x -> x+1}}{\texttt{f} \Rightarrow \texttt{fun x -> x+1}} \qquad 16+1 \Rightarrow 17}{\texttt{f 16} \Rightarrow 17} \qquad \dfrac{\dfrac{\texttt{s = fun y -> y*y}}{\texttt{s} \Rightarrow \texttt{fun y -> y*y}} \qquad 2*2 \Rightarrow 4}{\texttt{s 2} \Rightarrow 4} \qquad 17+4 \Rightarrow 21$$

$$\texttt{f 16 + s 2} \Rightarrow 21$$

// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen

327

Beispiel 3

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

Behauptung: `app (1::[]) (2::[]) ⇒ 1::2::[]`

Beweis

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{app = fun x y -> ...}}}{\texttt{app = fun x y -> ...}}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}} \quad \dfrac{\dfrac{\dfrac{}{\texttt{2::[]} \Rightarrow \texttt{2::[]}}}{\texttt{match [] ...} \Rightarrow \texttt{2::[]}}}{\texttt{app [] (2::[])} \Rightarrow \texttt{2::[]}}}{\texttt{1 :: app [] (2::[])} \Rightarrow \texttt{1::2::[]}}}{\texttt{match 1::[] ...} \Rightarrow \texttt{1::2::[]}}}{\texttt{app (1::[]) (2::[])} \Rightarrow \texttt{1::2::[]}}$$

// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen

Beweis

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{app = fun x y -> ...}}}{\texttt{app = fun x y -> ...}}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}} \quad \dfrac{\dfrac{\dfrac{}{\texttt{2::[]} \Rightarrow \texttt{2::[]}}}{\texttt{match [] ...} \Rightarrow \texttt{2::[]}}}{\texttt{app [] (2::[])} \Rightarrow \texttt{2::[]}}}{\texttt{1 :: app [] (2::[])} \Rightarrow \texttt{1::2::[]}}}{\texttt{match 1::[] ...} \Rightarrow \texttt{1::2::[]}}}{\texttt{app (1::[]) (2::[])} \Rightarrow \texttt{1::2::[]}}$$

// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen

Beweis

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{app = fun x y -> ...}}}{\texttt{app = fun x y -> ...}}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}} \quad \dfrac{\dfrac{\dfrac{}{\texttt{2::[]} \Rightarrow \texttt{2::[]}}}{\texttt{match [] ...} \Rightarrow \texttt{2::[]}}}{\texttt{app [] (2::[])} \Rightarrow \texttt{2::[]}}}{\texttt{1 :: app [] (2::[])} \Rightarrow \texttt{1::2::[]}}}{\texttt{match 1::[] ...} \Rightarrow \texttt{1::2::[]}}}{\texttt{app (1::[]) (2::[])} \Rightarrow \texttt{1::2::[]}}$$

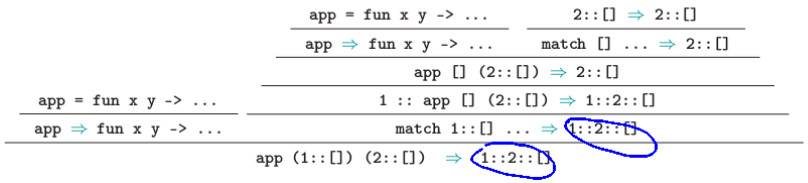
// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen

Beispiel 3

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

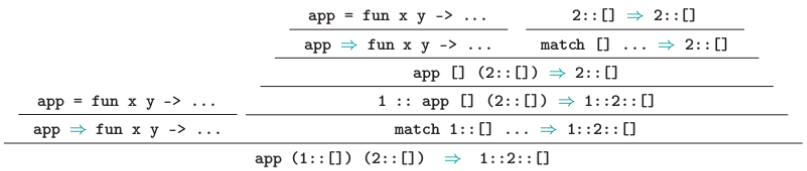
Behauptung: `app (1::[]) (2::[])` $\Rightarrow$ `1::2::[]`

Beweis



// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen

Beweis



// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen

Diskussion

- Die Big-Step operationelle Semantik ist nicht sehr gut geeignet, um Schritt für Schritt nachzu vollziehen, was ein MiniOcaml-Programm macht.
- Wir können damit aber sehr gut nachweisen, dass die Auswertung eine Funktion für bestimmte Argumentwerte stets terminiert:

  Dazu muss nur nachgewiesen werden, dass es jeweils einen Wert gibt, zu dem die entsprechende Funktionsanwendung ausgewertet werden kann ...

Diskussion

- Die Big-Step operationelle Semantik ist nicht sehr gut geeignet, um Schritt für Schritt nachzu vollziehen, was ein MiniOcaml-Programm macht.
- Wir können damit aber sehr gut nachweisen, dass die Auswertung eine Funktion für bestimmte Argumentwerte stets terminiert:

  Dazu muss nur nachgewiesen werden, dass es jeweils einen Wert gibt, zu dem die entsprechende Funktionsanwendung ausgewertet werden kann ...

Beispiel 3

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

Behauptung: `app (1::[]) (2::[])` $\Rightarrow$ `1::2::[]`

Beispiel-Behauptung

app l_1 l_2 terminiert für alle Listen-Werte l_1, l_2.

Beweis

Induktion nach der Länge n der Liste l_1.

$n = 0$: D.h. l_1 = []. Dann gilt:

$$\frac{\texttt{app = fun x y -> } \cdots \qquad \texttt{app} \Rightarrow \texttt{fun x y -> } \cdots \qquad \texttt{match [] with [] -> } l_2 \texttt{ | ...} \Rightarrow l_2}{\texttt{app [] } l_2 \Rightarrow l_2}$$

Beispiel-Behauptung

app l_1 l_2 terminiert für alle Listen-Werte l_1, l_2.

Beweis

Induktion nach der Länge n der Liste l_1.

$n = 0$: D.h. l_1 = []. Dann gilt:

$$\frac{\texttt{app = fun x y -> } \cdots \qquad \texttt{app} \Rightarrow \texttt{fun x y -> } \cdots \qquad \texttt{match [] with [] -> } l_2 \texttt{ | ...} \Rightarrow l_2}{\texttt{app [] } l_2 \Rightarrow l_2}$$

$n > 0:$ D.h. l_1 = `h::t`.

Insbesondere nehmen wir an, dass die Behauptung bereits für alle kürzeren Listen gilt. Deshalb haben wir:

$$\texttt{app t } l_2 \Rightarrow l$$

für ein geeignetes l. Wir schließen:

$$\dfrac{\texttt{app = fun x y -> ...} \qquad \dfrac{\dfrac{\texttt{app t } l_2 \Rightarrow l}{\texttt{h :: app t } l_2 \Rightarrow \texttt{h :: } l}}{\texttt{match h::t with } \cdots \Rightarrow \texttt{h :: } l}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}}$$

$$\texttt{app (h::t) } l_2 \Rightarrow \texttt{h :: } l$$

Diskussion

- Die Big-Step operationelle Semantik ist nicht sehr gut geeignet, um Schritt für Schritt nachzu vollziehen, was ein MiniOcaml-Programm macht.
- Wir können damit aber sehr gut nachweisen, dass die Auswertung eine Funktion für bestimmte Argumentwerte stets terminiert:

  Dazu muss nur nachgewiesen werden, dass es jeweils einen Wert gibt, zu dem die entsprechende Funktionsanwendung ausgewertet werden kann ...

$n > 0:$ D.h. l_1 = `h::t`.

Insbesondere nehmen wir an, dass die Behauptung bereits für alle kürzeren Listen gilt. Deshalb haben wir:

$$\texttt{app t } l_2 \Rightarrow l$$

für ein geeignetes l. Wir schließen:

$$\dfrac{\texttt{app = fun x y -> ...} \qquad \dfrac{\dfrac{\texttt{app t } l_2 \Rightarrow l}{\texttt{h :: app t } l_2 \Rightarrow \texttt{h :: } l}}{\texttt{match h::t with } \cdots \Rightarrow \texttt{h :: } l}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}}$$

$$\texttt{app (h::t) } l_2 \Rightarrow \texttt{h :: } l$$

332

Diskussion (Forts.)

- Wir können mit der Big-step-Semantik auch überprüfen, dass optimierende Transformationen korrekt sind.
- Schließlich können wir sie benutzen, um die Korrektheit von Aussagen über funktionale Programme zu beweisen !
- Die Big-Step operationelle Semantik legt dabei nahe, Ausdrücke als Beschreibungen von Werten aufzufassen.
- Ausdrücke, die sich zu den gleichen Werten auswerten, sollten deshalb austauschbar sein ...

Achtung

- Gleichheit zwischen Werten kann in MiniOcaml nur getestet werden, wenn diese keine Funktionen enthalten !!
- Solche Werte nennen wir vergleichbar. Sie haben die Form:

$$C \;::=\; \text{const} \mid (C_1, \ldots, C_k) \mid \texttt{[]} \mid C_1 :: C_2$$

Achtung

- Gleichheit zwischen Werten kann in MiniOcaml nur getestet werden, wenn diese keine Funktionen enthalten !!
- Solche Werte nennen wir vergleichbar. Sie haben die Form:

$$C \;::=\; \text{const} \mid (C_1, \ldots, C_k) \mid \texttt{[]} \mid C_1 :: C_2$$

- Offenbar ist ein MiniOcaml-Wert genau dann vergleichbar, wenn sein Typ funktionsfrei, d.h. einer der folgenden Typen ist:

$$c \;::=\; \texttt{bool} \mid \texttt{int} \mid \texttt{unit} \mid c_1 * \ldots * c_k \mid c\ \texttt{list}$$

Achtung

17 + 4
21

- Gleichheit zwischen Werten kann in MiniOcaml nur getestet werden, wenn diese keine Funktionen enthalten !!
- Solche Werte nennen wir vergleichbar. Sie haben die Form:

$$C \;::=\; \text{const} \mid (C_1, \ldots, C_k) \mid \texttt{[]} \mid C_1 :: C_2$$

- Offenbar ist ein MiniOcaml-Wert genau dann vergleichbar, wenn sein Typ funktionsfrei, d.h. einer der folgenden Typen ist:

$$c \;::=\; \texttt{bool} \mid \texttt{int} \mid \texttt{unit} \mid c_1 * \ldots * c_k \mid c\ \texttt{list}$$

Diskussion

- In Programmoptimierungen möchten wir gelegentlich Funktionen austauschen, z.B.

```
comp (map f) (map g) = map (comp f g)
```

- Offenbar stehen rechts und links des Gleichheitszeichens Funktionen, deren Gleichheit Ocaml nicht überprüfen kann

$\Longrightarrow$

Die Logik benötigt einen stärkeren Gleichheitsbegriff!

Erweiterung der Gleichheit

Wir erweitern die Ocaml-Gleichheit $=$ auf Werten auf Ausdrücke, die nicht terminieren, und Funktionen.

Nichtterminierung

$$\frac{e_1, e_2 \quad \text{terminieren beide nicht}}{e_1 = e_2}$$

Terminierung

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2 \qquad v_1 = v_2}{e_1 = e_2}$$

Strukturierte Werte

$$\frac{v_1 = v_1' \ \ldots \ v_k = v_k'}{(v_1, \ldots, v_k) = (v_1', \ldots, v_k')}$$

$$\frac{v_1 = v_1' \qquad v_2 = v_2'}{v_1 :: v_2 = v_1' :: v_2'}$$

Funktionen

$$\frac{e_1[v/x_1] = e_2[v/x_2] \quad \text{für alle} \quad v}{\texttt{fun}\ x_1\ \texttt{->}\ e_1 \ = \ \texttt{fun}\ x_2\ \texttt{->}\ e_2}$$

$\Longrightarrow$ extensionale Gleichheit

Wir haben:

$$\frac{e \Rightarrow v}{e = v}$$

Seien der Typ von e_1, e_2 funktionsfrei. Dann gilt:

$$\frac{e_1 = e_2 \qquad e_1 \quad \text{terminiert}}{e_1 = e_2 \ \Rightarrow \ \texttt{true}}$$

$$\frac{e_1 = e_2 \ \Rightarrow \ \texttt{true}}{e_1 = e_2 \qquad e_i \quad \text{terminieren}}$$

Das entscheidende Hilfsmittel für unsere Beweise ist das ...

Wir haben:

$$\frac{e \Rightarrow v}{e = v}$$

Seien der Typ von e_1, e_2 funktionsfrei. Dann gilt:

$$\frac{e_1 = e_2 \qquad e_1 \quad \text{terminiert}}{e_1 = e_2 \ \Rightarrow \ \texttt{true}}$$

$$\frac{e_1 = e_2 \ \Rightarrow \ \texttt{true}}{e_1 = e_2 \qquad e_i \quad \text{terminieren}}$$

Das entscheidende Hilfsmittel für unsere Beweise ist das ...

Substitutionslemma

$$\frac{e_1 = e_2}{e[e_1/x] = e[e_2/x]}$$

Wir folgern für funktionsfreie Ausdrücke e:

$$\frac{e_1 = e_2 \qquad e[e_1/x] \text{ terminiert}}{e[e_1/x] = e[e_2/x] \;\Rightarrow\; \texttt{true}}$$

Diskussion

- Das Lemma besagt damit, dass wir in jedem Kontext alle Vorkommen eines Ausdrucks e_1 durch einen Ausdruck e_2 ersetzen können, sofern e_1 und e_2 die selben Werte representieren.
- Das Lemma lässt sich mit Induktion über die Tiefe der benötigten Herleitungen zeigen (was wir uns sparen)
- Der Austausch von als gleich erwiesenen Ausdrücken gestattet uns, die Äquivalenz von Ausdrücken zu beweisen ...

Zuerst verschaffen wir uns ein Repertoir von Umformungsregeln, die die Gleichheit von Ausdrücken auf Gleichheiten anderer, möglicherweise einfacherer Ausdrücke zurück führt ...

Vereinfachung lokaler Definitionen

$$\frac{e_1 \text{ terminiert}}{\texttt{let } x = e_1 \texttt{ in } e \;=\; e[e_1/x]}$$

Zuerst verschaffen wir uns ein Repertoir von Umformungsregeln, die die Gleichheit von Ausdrücken auf Gleichheiten anderer, möglicherweise einfacherer Ausdrücke zurück führt ...

Vereinfachung lokaler Definitionen

$$\frac{e_1 \text{ terminiert}}{\texttt{let } x = e_1 \texttt{ in } e \;=\; e[e_1/x]}$$

Vereinfachung von Funktionsaufrufen

$$\frac{e_0 = \texttt{fun } x \texttt{ -> } e \qquad e_1 \text{ terminiert}}{e_0\, e_1 = e[e_1/x]}$$

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

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

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
                  | h::t -> h :: app t 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
```


$n > 0$: 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
```


7.3 Beweise für MiniOcaml-Programme

Beispiel 1

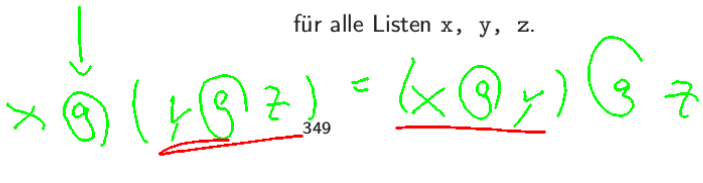
```
let rec app = fun x -> fun y -> match x
                  with [] -> y
                  | h::t -> h :: app t y
```

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`.



7.3 Beweise für MiniOcaml-Programme

Beispiel 1

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

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
```


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 [] -> app y z
                        | 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:

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

Behauptung

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

Beispiel 2

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