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:

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

:-)

$n > 0:$ 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:

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

:-)

Diskussion (Forts.):

- Wir können mit der Bigstep-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:

$$C \quad ::= \quad \text{const} \quad | \quad (C_1, \ldots, C_k) \quad | \quad [] \quad | \quad C_1 :: C_2$$

Achtung:

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

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

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

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

:-)

Achtung:

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

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

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

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

:-)

Für Ausdrücke e_1, e_2, e mit funktionsfreien Typen können wir Schlussregeln angeben ...

Substitutionslemma:

$$\frac{e_1 \Rightarrow v' \qquad e_2 \Rightarrow v' \qquad e[e_1/x] \Rightarrow v}{e[e_2/x] \Rightarrow v}$$

Beachte:

$$e_1 = e_2 \Rightarrow \texttt{true} \quad \Leftrightarrow \quad e_1 \Rightarrow v' \ \wedge \ e_2 \Rightarrow v' \quad \text{für ein } v' \quad \text{:-)}$$

Wir folgern:

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

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 liefern :-)
- Das Lemma lässt sich mit Induktion über die Tiefe der benötigten Herleitungen zeigen (was wir uns sparen :-))
- Der Austausch von als gleich erwiesenen Ausdrücken ist die Grundlage unserer Methode zum Nachweis der Äquivalenz von Ausdrücken ...

6.3 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 [] []
           =   []
           =   x      :-)
```

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

Wir schließen:

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

Diskussion:

- Bei den Gleichheitsumformungen haben wir einfache Zwischenschritte weggelassen :-)
- Eine Aussage: `exp1 = exp2`
  steht für: `exp1 = exp2` $\Rightarrow$ `true` und schließt damit die Terminierung der Auswertungen von `exp1, exp2` ein.
- Zur Korrektheit unserer Induktionsbeweise benötigen wir, dass sämtliche 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 [] y
               =  y
               =  rev1 [] y
               =  rev1 x y      :-)
```

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

Wir schließen:

```
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`.

$\boxed{n = 0:}$ 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 :-)