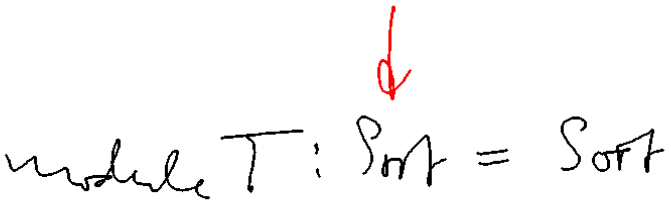
Script generated by TTT

Title: Seidl: Info2 (12.01.2018)

Date: Fri Jan 12 08:33:04 CET 2018

Duration: 93:04 min

Pages: 26



Ein MiniOcaml-Programm ...

```
let rec comp = fun f g x -> f (g x)
    and map  = fun f list -> match list
               with [] -> []
               | x::xs -> f x :: map f xs
```

Beispiele für Werte ...

```
1
(1, [true; false])
fun x -> 1 + 1
[fun x -> x+1; fun x -> x+2; fun x -> x+3]
```


Ein MiniOcaml-Programm ...

```
let rec comp = fun f g x -> f (g x)
    and map  = fun f list -> match list
               with [] -> []
               | x::xs -> f x :: map f xs
```

Beispiele für Werte ...

```
1
(1, [true; false])
fun x -> 1 + 1
[fun x -> x+1; fun x -> x+2; fun x -> x+3]
```

317

Idee

- Wir definieren eine Relation: $e \Rightarrow v$ zwischen Ausdrücken und ihren Werten $\Longrightarrow$ Big-Step operationelle Semantik.
- Diese Relation definieren wir mit Hilfe von Axiomen und Regeln, die sich an der Struktur von e orientieren.
- Offenbar gilt stets: $v \Rightarrow v$ für jeden Wert v.

Tupel

$$\frac{e_1 \Rightarrow v_1 \quad \ldots \quad e_k \Rightarrow v_k}{(e_1, \ldots, e_k) \Rightarrow (v_1, \ldots, v_k)}$$

Listen

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 :: e_2 \Rightarrow v_1 :: v_2}$$

Globale Definitionen

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

Lokale Definitionen

$$\frac{e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{\texttt{let } x = e_1 \texttt{ in } e_0 \Rightarrow v_0}$$

Funktionsaufrufe

$$\frac{e \Rightarrow \texttt{fun } x \texttt{ -> } e_0 \quad e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{e\ e_1 \Rightarrow v_0}$$

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

$$\frac{e_0 \Rightarrow \texttt{fun } x_1 \ldots x_k \texttt{ -> } e \quad e_1 \Rightarrow v_1 \ \ldots \ e_k \Rightarrow v_k \quad e[v_1/x_1, \ldots, v_k/x_k] \Rightarrow v}{e_0\ e_1 \ \ldots \ e_k \Rightarrow v}$$

Diese abgeleitete Regel macht Beweise etwas weniger umständlich.

Pattern Matching

$$\frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1,\ldots,v_k/x_k] \qquad e_i[v_1/x_1,\ldots,v_k/x_k] \Rightarrow v}{\texttt{match}\ e_0\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_m\ \texttt{->}\ e_m \Rightarrow v}$$

— sofern v' auf keines der Muster $p_1,\ldots,p_{i-1}$ passt ;-)

4 + 7 ⇒ 11

Eingebaute Operatoren

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2 \qquad v_1\ \mathrm{op}\ v_2 \Rightarrow v}{e_1\ \mathrm{op}\ e_2 \Rightarrow v}$$

Die unären Operatoren behandeln wir analog.

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

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

Beispiel 2

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

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

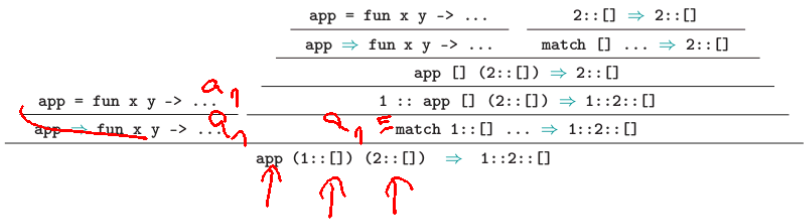
// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen

Beispiel 3

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

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

Beweis



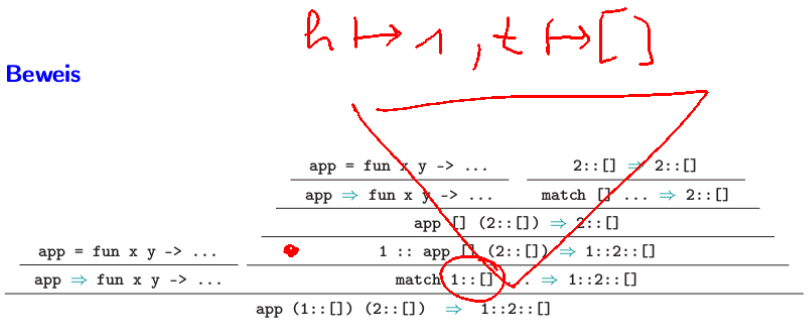
// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen

Beispiel 3

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

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

Beweis



// 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::[]) ⇒ 1::2::[]`

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

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

// 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::[]`

325

Beweis

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

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

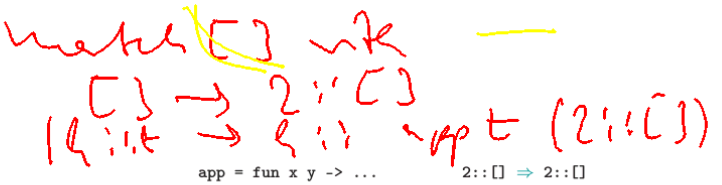
326

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



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

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

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 ...

Beweis

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

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

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-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:

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

$n > 0:$ 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:

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