Ein MiniOcaml-Programm ...

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

Ein MiniOcaml-Programm ...

```
let rec comp = fun f -> fun g -> fun x -> f (g x)
    and map  = fun f -> fun 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]
```

Idee 1:

- Wir definieren einen Schritt der Berechnung als eine Relation: $e \rightarrow e'$ zwischen Ausdrücken.
- Eine Berechnung ist eine Folge von Berechnungsschritten

  $\Longrightarrow$ Small-Step operationelle Semantik
- Auf Werten ist kein weiterer Berechnungsschritt möglich, d.h. bei Erreichen eines Wert endet eine Berechnung :-)
- Einzelne Berechnungsschritte definieren wir mit Hilfe geeigneter Axiome und Regeln ...

  // $v, v_1, \ldots$ stehen für Werte,
  // $e, e_1, \ldots$ bezeichnen beliebige Ausdrücke

Tupel:

$$\frac{e_i \;\to\; e_i'}{(v_1, \ldots, v_{i-1}, e_i, \ldots, e_k) \;\to\; (v_1, \ldots, v_{i-1}, e_i', e_{i+1}, \ldots, e_k)}$$

Listen:

$$\frac{e_1 \;\to\; e_1'}{e_1 :: e_2 \;\to\; e_1' :: e_2} \qquad \frac{e_2 \;\to\; e_2'}{v_1 :: e_2 \;\to\; v_1 :: e_2'}$$

Globale Definitionen:

$$\frac{f = e}{f \;\to\; e}$$

Lokale Definitionen:

$$\frac{e_1 \longrightarrow e_1'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_0 \longrightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_0}$$

$$\texttt{let}\ x = v\ \texttt{in}\ e_0 \longrightarrow e_0[v/x]$$

Funktionsaufrufe:

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'}$$

$$(\texttt{fun}\ x \texttt{->}\ e_0)\ v \longrightarrow e_0[v/x]$$

Pattern Matching:

$$\frac{e_0 \;\longrightarrow\; e_0'}{\texttt{match}\; e_0 \;\texttt{with}\; \mathit{plist} \;\longrightarrow\; \texttt{match}\; e_0' \;\texttt{with}\; \mathit{plist}}$$

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

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

// "$\equiv$" bezeichnet syntaktische Gleichheit

Eingebaute Operatoren:

$$\frac{e_1 \longrightarrow e_1'}{e_1 \;\mathsf{op}\; e_2 \longrightarrow e_1' \;\mathsf{op}\; e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v \;\mathsf{op}\; e_2 \longrightarrow v \;\mathsf{op}\; e_2'}$$

$$\frac{v_1 \;\mathsf{op}\; v_2 = v}{v_1 \;\mathsf{op}\; v_2 \longrightarrow v}$$

Die unären Operatoren behandeln wir analog :-)

Beispiel:

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

$$
\begin{aligned}
\texttt{f 16 + s 2} &\longrightarrow \texttt{(fun x -> x+1) 16 + s 2} \\
&\longrightarrow \texttt{(16+1) + s 2} \\
&\longrightarrow \texttt{17 + s 2} \\
&\longrightarrow \texttt{17 + (fun y -> y*y) 2} \\
&\longrightarrow \texttt{17 + (2*2)} \\
&\longrightarrow \texttt{17 + 4} \\
&\longrightarrow \texttt{21}
\end{aligned}
$$

Beispiel:

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

Frage:

Welchen Wert liefert `app (1::[]) (2::[])` ?

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

$\longrightarrow$ `(fun x -> fun y -> ...) (1::[]) (2::[])`

$\longrightarrow$ `(fun y -> match ...) (2::[])`

$\longrightarrow$ `match 1::[] with ...`

$\longrightarrow$ `1 :: app [] (2::[])`

$\longrightarrow$ `1 :: (fun x -> fun y -> ...) [] (2::[])`

$\longrightarrow$ `1 :: (fun y -> match ...) (2::[])`

$\longrightarrow$ `1 :: match [] with ...`

$\longrightarrow$ `1 :: 2 :: []`

Diskussion:

- Die Small-Step operationelle Semantik ist gut geeignet, um eine einzelne Berechnung nach zu vollziehen :-)
- Sie ist weniger gut geeignet, um nachzuweisen, dass verschiedene Ausdrücke den selben Wert liefern :-(

Diskussion:

- Die Small-Step operationelle Semantik ist gut geeignet, um eine einzelne Berechnung nach zu vollziehen :-)
- Sie ist weniger gut geeignet, um nachzuweisen, dass verschiedene Ausdrücke den selben Wert liefern :-(

Idee 2:

- Wir definieren eine Relation: $e \Rightarrow v$ zwischen Ausdrücken und ihren Werten $\Longrightarrow$ Big-Step operationelle Semantik.
- Diese Relation definieren wir erneut 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:

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

Listen:

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

Globale Definitionen:

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

Lokale Definitionen:

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

Funktionsaufrufe:

$$\frac{e_1 \Rightarrow \texttt{fun}\ x\ \texttt{->}\ e_0 \qquad e_2 \Rightarrow v_2 \qquad e_0[v_2/x] \Rightarrow v_0}{e_1\ e_2 \Rightarrow v_0}$$

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 ;-)

Eingebaute Operatoren:

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2 \qquad v_1 \,\mathsf{op}\, v_2 = v}{e_1 \,\mathsf{op}\, e_2 \Rightarrow v}$$

Die unären Operatoren behandeln wir analog :-)

Beispiel:

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

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

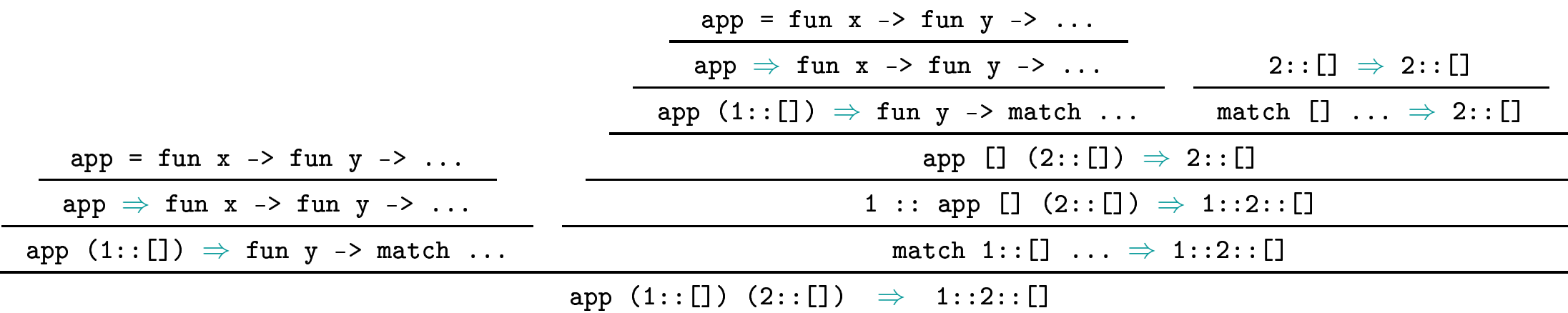
// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen :-)

Beispiel:

```
let rec app = fun x -> fun 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 :-)

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 überprüfen, dass optimierende Transformationen korrekt sind :-)
- Schließlich können wir sie benutzen, um die Korrektheit von Aussagen über funktionale Programme zu beweisen !
- Zuerst müssen wir aber zeigen, dass sie zur Small-Step operationellen Semantik äquivalent ist ...

Notation:

$e_1 \to^* e_2$ heißt $e_1 \equiv e_2$ (textuelle Gleichheit) oder es gibt eine Folge von Berechnungsschritten: $e_1 \to \ldots \to e_2$

Satz

Für jedes MiniOcaml-Programm gilt:

$$e \to^* v \quad \text{genau dann wenn} \quad e \Rightarrow v$$

// e Ausdruck, v Wert

Diskussion:

- Der Satz lässt sich mit Hilfe von Induktion beweisen einmal nach der Länge von Folgen von Berechnungsschritten, einmal nach der Tiefe von Ableitungen der Big-Step operationellen Semantik :-)
- Die Big-Step operationelle Semantik legt nahe, Ausdrücke als Beschreibungen von Werten aufzufassen.
- Ausdrücke, die sich zu den gleichen Werten auswerten, sollten deshalb austauschbar sein ...